VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Analýza a transformace XML dokumentu

Analysis and transformation of an input XML document

2010 Bc. Rostislav Šuta

Zadání

Téma:

Analýza a transformace XML dokumentu

Zásady pro vypracování:

Hlavním cílem této práce je vytvoření nástroje pro analýzu a jednoduchou transformaci vstupních XML dokumentů. Nástroj bude uživateli umožňovat ovlivnit výslednou transformaci na základě statistik XML dokumentu.

1. Implementace nástroje pro analýzu vstupního XML dokumentu.
2. Prezentace statistik v uživatelsky přívětivé formě. Uživatel zvolí parametry transformace na základě těchto statistik.
3. Transformace XML dokumentu na základě parametru zvolených uživatelem.
4. Testování efektivity transformace a porovnání její rychlosti s XLST transformací pomocí SAXON.

Vedoucí:

Ing. Radim Bača, Ph.D.

Obor:

2612T025 Informatika a výpočetní technika

Akademický rok:

2009/2010

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2010 .....

Abstrakt

Cílem této práce je implementace nástroje pro analýzu a jednoduchou transformaci XML dokumentů za účelem zrychlení vyhledávání dat v těchto dokumentech. Nástroj bude umožňovat účinně analyzovat vstupní XML dokument a výsledky analýzy zpracovávat do databáze. Tyto statistiky budou poté v rozumné formě prezentovány uživateli, který svým výběrem ovlivní výslednou transformaci XML dokumentu. Transformace XML dokumentů bude poté porovnávána z hlediska efektivity a rychlosti. K porovnání poslouží XML dokument vytvořený XSLT transformací pomocí XSLT procesoru Saxon. Navíc je v této práci obsaženo i porovnání při vyhledávání dat pomocí dotazů nad těmito transformovanými dokumenty. Výsledky těchto porovnání jsou zároveň součástí této práce. Samotná textová část této diplomové práce je logicky členěna na část teoretickou, implementační a testovací. Závěr obsahuje shrnutí všech nabytých poznatků a výsledků.

Klíčová slova: Diplomová práce, XML dokument, transformace, analýza, Saxon, XMark, databáze

Abstract

The aim of this thesis is to implement tool for analysis and simple transformation of XML documents leading to the faster searching of data in these documents. The tool will provide effective analysis of the input XML document and results of this analysis will be processed into the database. These statistics will be presented in a reasonable form to user who will affect the final transformation of the XML document. Then, transformation of XML documents will be compared according to efficiency and speed. As comparing material will be used document created by XSLT transformation using Saxon processor. Moreover, in this work is included a comparison in searching data in these documents using data queries. The results of these comparisons are also included in this work. Text part of this thesis is logically structured in theoretical part, implementation and testing. The conclusion contains a summary of all the acquired knowledge and results.

Keywords: Thesis, XML document, transformation, analysis, Saxon, XMark, database

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
DOM	–	(Document Object Model
DTD	–	Document Type Definition
SAX	–	(Simple API for XML
W3C	–	World Wide Web Consortium
XPath	–	XML Path Language
XPointer	–	XML Pointer Language
XLink	–	XML Linking Language
XML	–	Extensible Markup Language
XSL	–	Extensible Stylesheet Language
XSLT	–	Extensible Stylesheet Language Transformations

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Tato diplomová práce se zabývá problematikou transformace XML dokumentů za účelem zrychlení a zefektivnění vyhledávání dat v těchto dokumentech. Jedním z hlavních problémů při práci s rozsáhlými XML dokumenty je relativně zdlouhavé vyhledávání dat při užití dotazů na konkrétní hodnoty. Jednou z možností řešení tohoto problému je vhodná optimalizace struktury XML dokumentu do podoby, která umožní rychlejší a efektivnější vyhledávání nad těmito daty. Cílem této práce je vytvořit nástroj pro analýzu a jednoduchou transformaci těchto XML dokumentů. Nástroj má umožnit účinnou analýzu vstupního XML dokumentu podle požadovaných vlastností a výsledky této analýzy uložit pro další zpracování. Uživatel poté bude moci na základě předložených statistik ovlivnit výslednou transformaci XML dokumentu. Součástí práce je i testování efektivity transformace takto nově vytvořených dokumentů a porovnávání výsledku s jinými aplikacemi.

Práce je logicky členěna, na na úvod je objasněna podstata XML dokumentů a principy transformace těchto dokumentů pomocí šablon. Na tuto část navazuje kapitola popisující návrh aplikace pro transformaci a rozbor její funkčnosti. Dále se práce zabývá jednotlivými fázemi transformace dokumentu a jejich řešeními ve vytvořeném programu. Poté následuje testovací část, kde jsou popsány jednotlivé testy a jejich výsledky, a práce je zakončena závěrečným shrnutím.

2 Technologie XML

2.1 Úvod do XML

Extensible Markup Language (zkráceně XML) je obecný značkovací jazyk vyvinutý a standardizovaný konsorciem W3C. Umožňuje snadné vytváření konkrétních značkovacích jazyků pro libovolné účely a různé typy zdrojových dat. Jazyk XML je určen především pro výměnu dat mezi aplikacemi a pro publikaci dokumentů. Jazyk XML pouze popisuje strukturu z hlediska věcného obsahu jednotlivých částí, nezabývá se už ale vzhledem těchto dat. Zpracování jazyka XML je v dnešní době pro jeho popularitu podporováno řadou nástrojů a programovacích jazyků.[1, 2]

XML není prvním značkovacím jazykem, ale byl vytvořen podle již dříve navržených značkovacích jazyků. Značkovací jazyk je jazyk, jehož zdrojový text obsahuje současně jak vlastní text, tak instrukce pro jeho zpracování. Ty se u jazyka XML vyskytují v podobě značek (tagů). Typickým rysem značkovacích jazyků jsou znaky se speciálním významem. Tyto znaky slouží k vymezení řídicích konstrukcí - příkazů nebo značek. V XML mají speciální význam znaky „menší než“ (<) a „větší než“ (>), které zahajují a ukončují značky. Obsah mezi těmito značkami je chápán jako text pro zpracování.

Hlavní vlastnosti XML formátu jsou:

- XML je de facto standardem pro výměnu informací
- má mezinárodní podporu
- je možná snadná konverze do jiných formátů
- podporuje automatickou kontrolu struktury dokumentu
- vytváření jednoduchých anotací textu

2.2 Syntaxe jazyka XML

2.2.1 Elementy

Základní jednotkou XML dokumentu je element. Každý XML dokument se skládá z těchto elementů, které jsou do sebe navzájem vnořeny. Elementy se v textu ohraničují pomocí tzv. tagů. Většina elementů se skládá ze dvou tagů: počátečního a ukončovacího.

```
<kniha> Název knihy. </kniha>
```

Ukázka obsahuje jeden element `kniha`. Obsah tohoto elementu je vyznačen pomocí počátečního tagu `<kniha>` a koncového tagu `</kniha>`. Takto zapsaný text je nejjednodušší způsob vytvoření XML dokumentu. Názvy tagů se zapisují mezi znaky „<“ a „>“. Ukončovací tag má pro odlišení od počátečního před svým názvem znak „/“.

V každém XML dokumentu platí, že pro všechny počáteční tagy musí existovat odpovídající ukončovací tag. Počáteční tag může být zapsán jako element s prázdným obsahem. Pro srozumitelnost uvedu dva příklady špatného zápisu XML dokumentu. V prvním případě nebude existovat správný ukončovací tag a ve druhém případě budou tagy tzv. překříženy.

1. <kniha> <autor> </kniha>
2. <kniha> <autor> </kniha> </autor>

Specifikace XML definuje několik úrovní správnosti XML dokumentu. Pokud dokument splňuje základní syntaktická pravidla (např. párové tagy - viz výše), říkáme, že dokument je **správně strukturovaný** (angl. well-formed). Tímto označením tedy poukazujeme na dokument, který splňuje formální požadavky popsané ve specifikaci XML. Pokud navíc dokument obsahuje *DTD* (viz kapitola 2.2.4) nebo je na něj odkaz v deklaraci typu dokumentu, *parser* kontroluje, zda struktura dokumentu odpovídá definovanému DTD. Pokud ano, můžeme dokument označit jako **validní** (angl. valid). Pokud struktura dokumentu neodpovídá pravidlům definovaným v příslušném DTD, je dokument označen jako **nevalidní** (angl. non valid). Z výše uvedeného tedy platí, že i nevalidní dokument může být správně strukturovaný. Zároveň také platí, že pokud je dokument validní, je vždy správně strukturovaný. Funkce parseru je blíže vysvětlena v kapitole 3.1.

2.2.2 Atributy

Elementy jsou základní jednotkou každého dokumentu. U každého počátečního tagu lze definovat navíc atributy. Tyto atributy se obvykle používají k upřesnění významu elementu. V následujícím příkladu přiřadíme knize atribut `id` s hodnotou 1.

```
<kniha id="1"> Název knihy. </kniha>
```

Atributů může být i více, pak je stačí oddělit mezerou. Zápis by vypadal asi takto:

```
<kniha id="1" rok="2009"> Název knihy. </kniha>
```

2.2.3 Kořenový element

Jedním z požadavků na správně strukturovaný dokument je, že každý XML dokument musí být celý obsažen v jednom kořenovém elementu. V následujícím příkladu máme XML soubor reprezentující strukturu knih v knihovně. Model tohoto XML dokumentu lze vidět na straně 10 na obrázku 1. Vlevo je zobrazen XML dokument, který není uzavřen v kořenovém elementu a vpravo je zobrazen upravený, správně strukturovaný dokument. V tomto případě tedy stačí přidat kořenový element, který celou strukturu zapouzdří.

```
<Beletrie>                          <Knihovna>
    ...                                 <Beletrie>
    <Kniha>                                 ...
        <Název>...</Název>                  <Kniha>
        <Autor>...</Autor>                      <Název>...</Název>
        <Rok>...</Rok>                          <Autor>...</Autor>
    </Kniha>                                    <Rok>...</Rok>
</Beletrie>                                 </Kniha>
<Detektivky>                            </Beletrie>
    ...                                 <Detektivky>
</Detektivky>                               ...
<Naučné>                                </Detektivky>
    ...                                 <Naučné>
</Naučné>                                   ...
                                        </Naučné>
                                    </Knihovna>
```

Výpis 1: Úprava ne správně strukturovaného dokumentu

2.2.4 DTD

Jenda z výhod ukládání dokumentů ve formátu XML spočívá v možnosti zachytit pomocí elementů strukturu dokumentu. Při práci s XML dokumenty používáame parser. Parser je program, který kontroluje, zdali je XML dokument správně strukturovaný. Funkce parseru a jeho implementace jsou blíže popsány v kapitole 3.1.
DTD jsou poměrně starou a v mnoha ohledech nedostatečnou technologií (nepodporují jmenné prostory), ale i přesto se stále používají ve veliké míře. Tato obliba je dána výbornou podporou v různých aplikacích a parserech. V dnešní době existují i další a efektnější způsoby popisu struktury jako například XML schémata.

Pokud chceme využít DTD, musíme použité DTD upřesnit pomocí deklarace typu dokumentu - `DOCTYPE`. Tato deklarace se umísťuje na začátek dokumentu, hned za XML deklaraci. Obvykle je DTD uloženo v samostatném souboru, aby mohlo být opakovaně využito více dokumenty. V tomto případě má deklarace tvar:

```
<!DOCTYPE [kořenový element] SYSTEM "[URL]" >
```

`URL` udává adresu nebo jméno souboru, ve kterém je uloženo DTD. `Kořenový element` je jméno elementu, ve kterém bude obsažen celý dokument.

Na následujícím příkladu lze vidět jednoduchá DTD deklarace pro XML dokument. Strukturu XML dokumentu lze vidět na obrázku 1. Každý element je definován samostatně, pokud obsahuje další podelementy, jsou vypsány v závorce, pokud obsahuje text pro zpracování, je zde uvedeno „#PCDATA“.

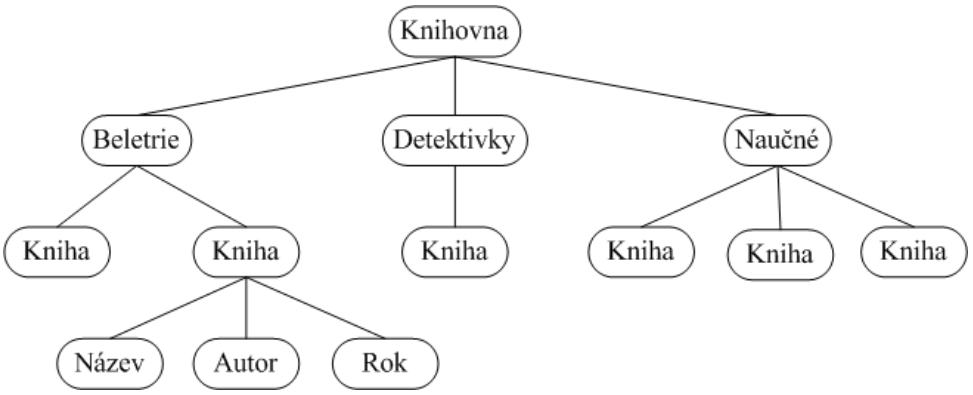
KNIHOVNA.DTD
_ _ _ _ _ _ _ _ _

```
<!ELEMENT Knihovna (Beletrie, Detektivky, Naučné)>
<!ELEMENT Beletrie (Kniha)>
<!ELEMENT Detektivky (Kniha)>
<!ELEMENT Naučné (Kniha)>
<!ELEMENT Kniha (Název, Autor, Rok)>
<!ELEMENT Název (#PCDATA)>
<!ELEMENT Autor (#PCDATA)>
<!ELEMENT Rok (#PCDATA)>
```

Výpis 2: DTD deklarace pro XML dokument

2.2.5 Model XML dokumentu

Dokument lze modelovat jako strom, kde uzel odpovídá elementu nebo atributu a hrana propojuje dva uzly, pokud mají vztah rodič - dítě. V této reprezentaci XML stromu existuje pro každý uzel *značkovaná cesta*. Značkovaná cesta nám v dokumentu udává cestu od kořene k elementu. Jednotlivé elementy jsou pak odděleny znalem „/“. Například pro autory knih v Beletrii nám podle následujícího obrázku vznikne značkovaná cesta: `/Knihovna/Beletrie/Kniha/Autor`. Následující obrázek slouží také jako model XML dokumentu k příkladům v předchozích kapitolách.



Obrázek 1: Model XML dokumentu

3 Nástroje pro práci s XML daty

3.1 Přístupy ke zpracování XML dat

Při zpracování XML dokumentu a získání jeho obsahu můžeme použít dva druhy parserů. První typ parseru jsou přímo spustitelné soubory, druhý typ jsou parsery v podobě knihoven, které můžeme použít v programech. Výhoda druhého řešení při programování je, že nemusíme kontrolovat syntaxi XML dokumentu. Tu automaticky zkontroluje parser, navíc může ověřit i validitu proti danému DTD nebo XML schématu. Více o DTD se můžete dovědět v kapitole 2.2.4. Při použití pak aplikace nemusí obsahovat tolik kódu pro ošetření chyb ve zpracovávaných datech.
Další výhodou parseru je, že obsah dokumentu zpřístupní v programátorsky „příjemné podobě“. Pro práci s XML dokumentem existují standardizovaná rozhraní tzv. API. V dnešní době se používají dvě rozhraní pro přístup k XML datům: DOM a SAX. Tato dvě rozhraní se liší především v přístupu ke zpracovávanému XML dokumentu a práci s jednotlivými objekty. Tyto parsery existují pro většinu běžně používaných programovacích jazyků jako jsou Java, C++ apod.

SAX

Rozhraní SAX (Simple API for XML) je založeno na řízení pomocí událostí. Pomocí rozhraní vytvoříme vazbu mezi událostmi generovaných parserem a programovým kódem. V praxi to znamená, že si definujeme funkce volající se v okamžiku, kdy parser narazí na začátek elementu, obsah elementu, konec elementu, komentář apod. Naší funkci jsou pak parserem předány všechny potřebné parametry jako např. název elementu.
Výhoda událostmi řízeného přístupu je v jeho rychlosti a malé spotřebě paměti. Jednotlivé události jsou vyvolávány postupně, jak je čten dokument. Naproti tomu rozhraní DOM vyžaduje načtení celého dokumentu předtím, než s ním začne pracovat. Pokud tedy nepotřebujeme funkčnost DOM přístupu, vyplatí se z hlediska rychlosti a paměti použít SAX.
Rozhraní SAX dnes podporuje velké množství parserů, i když samotné rozhraní není definováno pomocí žádného standardu konsorcia W3C nebo jiné standardizační organizace. Rozhraní vzniklo společným úsilím vývojářů skupiny xml-dev a představuje de facto standard.

Při návrhu programu pro tuto diplomovou práci jsem použil kvůli rychlosti čtení vstupního XML dokumentu a práci s načítanými daty právě tuto možnost přístupu k XML datům. V prostředí Java jsou tyto funkce dostupné prostřednictvím balíku `org.xml.sax`.

DOM

Rozhraní DOM (Document Object Model) je postaveno na úplně odlišném principu než SAX. Dokument je zde reprezentován jako stromová hierarchická struktura, kde každému elementu odpovídá jeden uzel stromu. Odpovídající uzly mají instrukce pro zpracování

apod. Tomuto způsobu reprezentace se říká grove (Graph Representation Of property ValuEs). Rozhraní DOM obsahuje funkce, které nám umožňují celý strom dokumentu procházet, modifikovat jeho jednotlivé uzly, mazat je a přidávat. Na rozdíl od SAXu nemusíme dokument procházet od začátku do konce, ale můžeme se v něm pohybovat dle potřeby. Z tohoto důvodu má rozhraní DOM uplatnění především v aplikacích, které provádějí náročnější operace s dokumentem jako jsou například editory, prohlížeče nebo formátovače. Ukázku stromové struktury XML dokumentu lze vidět na obrázku 1 na straně 10.

Rozhraní DOM je standardem konsorcia W3C. Původně byl DOM vytvořen především proto, aby nové verze prohlížečů podporující XML používaly stejný objektový model pro přístup k dokumentu ze skriptových jazyků jako je např. JavaScript. Bez tohoto standardu by nebyla taková možnost kompatibility internetových prohlížečů. Rozhraní DOM obsahuje například Internet Explorer nebo Mozilla Firefox.

3.2 Dotazovací jazyky

3.2.1 XPath

XPath je jednoduchý dotazovací jazyk, pomocí kterého lze adresovat části XML dokumentu. Pomocí tohoto jazyka můžeme z XML dokumentu vybírat jednotlivé elementy a pracovat s jejich hodnotami nebo atributy.[6]

Základní konstrukcí jazyka XPath je *cesta* k XML uzlu. Cesta nám v XML dokumentu určuje pozici konečného uzlu, se kterým pak můžeme dále pracovat. Tato cesta se skládá z jednoho nebo více *kroků*, přičemž každý z těchto kroků může obsahovat prvky:

- identifikátor osy XPath
- test uzlu
- podmínky

Povinná část každého kroku je pouze test uzlu, ostatní dva prvky jsou nepovinné. Jednotlivé kroky XPath výrazu se spojují znaky „/“, vyhodnocují se zleva doprava a následující krok pracuje s množinou určenou předchozím krokem. Výsledkem tohoto dotazu může být libovolný typ uzlu, element, atribut nebo i textový uzel.

Oddělování kroků

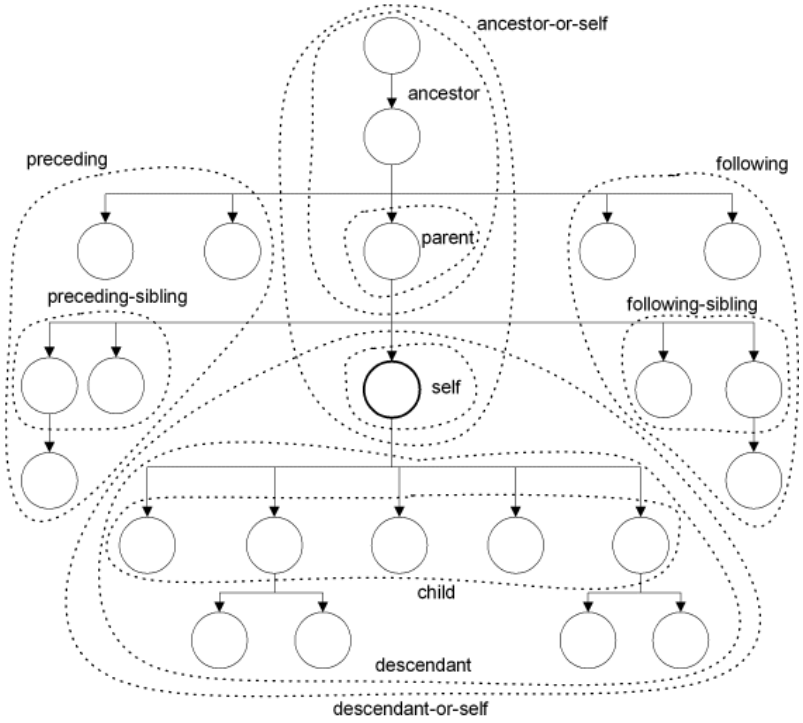
Symbol „/“ slouží k oddělování kroků v utvářené cestě. Pokud je tento znak na začátku cesty, tak tato cesta není vztažena k aktuálnímu elementu, ale počítá se od kořene dokumentu.
Symbol „//“ slouží k překonání víceúrovňové struktury. Pokud jsou tato dvě lomítka na začátku cesty, berou se při procházení struktury dokumentu v potaz všechny cesty obsahující kroky uvedené za těmito dvěma lomítky. Takto lze snadno vybrat libovolný element z kteréhokoliv místa v dokumentu, nemusíme vypisovat celou cestu k elementu.

Identifikátory osy

Testu uzlu muže předcházet takzvaný identifikátor osy. Ten určuje směr procházení XML dokumentu, to znamená odkud se uzly k vyhodnocení budou vybírat. Pokud identifikátor osy neuvedeme, použije se implicitní osa „`child::`“.
Možnosti identifikátoru jsou například:

- child:: - vyhodnocují se všichni přímí potomci aktuálního uzlu
- descendand:: - vyhodnocují se všichni potomci aktuálního uzlu
- ancestor:: - vyhodnocují se všichni předci aktuálního uzlu
- following:: - vyhodnocují se pouze následující uzly
- preceding:: - vyhodnocuje se pouze předchozí uzly

Přehlednou grafickou ukázku identifikátorů os můžete najít na obrázku 2.



Obrázek 2: Ukázka os XPath v XML dokumentu

Testy uzlu

Test uzlu dále vymezuje množinu uzlu, která byla určena identifikátorem osy.

- Uzel určený názvem - vybere všechny XML elementy s daným názvem. Zapíšeme jej jednoduchým zápisem testovaného názvu elementu.
- Uzel určený typem - zohledňuje typ uzlu a vybírá pouze určitý typ. Zapisuje se typem uzlu s prázdnými kulatými závorkami. Možnosti jsou: `comment()`, `text()`, `processing-instruction()` a `node()`.

Podmínky

Podmínky se zapisují do hranatých závorek. Tyto podmínky nám zužují výsledky předchozího vyhodnocení. Pokud místo podmínky napíšeme pouze číslo, vybere podmínka pouze uzel umístěný na pozici s daným pořadovým číslem.

```
identifikátor ::uzel[podminka]
```

Dotazy na hodnotu

Podmínky se v XPath výrazech používají například v dotazech na hodnotu daného uzlu. Pomocí podmínky můžeme omezit množinu uzlů, které budou odpovídat cílovému XPath dotazu. Podmínky můžeme aplikovat na hodnotu elementu nebo hodnotu jeho atributů. V této práci budeme využívat podmínek v XPath dotazech v testovací části, kde budeme měřit rychlost hledání a počet diskových přístupů na původních a transformovaných XML souborech. Pomocí podmínek budeme specifikovat množiny uzlů a ovlivňovat tak složitost hledání v XML dokumentu.

Ukážeme si jednoduchý příklad použití podmínky na struktuře XML dokumentu na obrázku 1 (strana 10). V prvním případě budeme chtít vybrat všechny knihy z Beletrie, kde bude autor „Karel Čapek“. Ve druhém případě si určeme, že element `Autor` má atribut `id`, které obsahuje číslo autora. Budeme také vybírat díla z Beletrie od Karla Čapka.

```
1. /Knihovna/Beletrie/Kniha[/Autor = 'Karel Čapek']
2. /Knihovna/Beletrie/Kniha/Autor[@id = '001']
```

Výše uvedené jsou základy jazyka XPath. Tento jazyk však obsahuje ještě mnoho operátorů a funkcí, jejichž popis je mimo rozsah této práce.

3.2.2 XQuery

XML se nepoužívá pouze pro přenos dat, ale i jako úložiště strukturovaných dat. Tato data je potřeba prohledávat, vybírat z nich dílčí údaje, počítat statistiky apod. Jazyk SQL zde nelze použít, protože datový model XML dokumentu je strom a XPath je příliš jednoduchý a mnoho věcí neumí. Proto byla potřeba vytvořit nový jazyk, který by tyto funkce dokázal. Tímto jazykem se stal XQuery.

Dotazovací jazyk XQuery obsahuje prvky jazyka XPath (tedy XPath je jakousi podmnožinou jazyka XQuery) a vychází také ze základů jazyka SQL. Pro dotazovací jazyk XQuery jsou charakteristické takzvané FLWOR výrazy. Ty jsou inspirovány jazykem SQL a slouží k získávání informací z dokumentu pomocí sady podmínek a omezení, kde pro tato omezení mohou být vybírány pomocí jazyka XPath příslušné uzly či struktury. Dalo by se napsat, že XQuery = XPath 2.0 + FLWOR výrazy + výrazy konstruující nové elementy + uživatelsky definované funkce.

Struktura FLWOR výrazu je následující:

- FOR - výběr posloupnosti uzlů k dalšímu zpracování
- LET - přiřazení proměnných pro každý prvek posloupnosti
- WHERE - filtrování uzlů v posloupnosti
- ORDER BY - seřazení vybraných a odfiltrovaných uzlů
- RETURN - specifikace výstupu pro každý vybraný a odfiltrovaný uzel

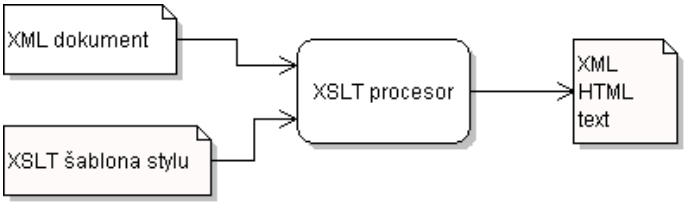
Výhodou XQuery je, že lze výrazy libovolně kombinovat dohromady a výsledek jednoho výrazu může sloužit jako parametr dalšího výrazu. Důležitým faktem je, že výsledkem XQuery dotazu je vždy instance datového modelu. XQuery také obsahuje silnou typovou kontrolu, takže je zajištěna větší bezpečnost.

3.3 Stylový jazyk XSLT

3.3.1 Základní princip XSLT

Základní myšlenkou, na které staví většina značkovacích jazyků včetně XML, je oddělení obsahu dokumentu od jeho vzhledu. Značky použité v XML dokumentu označují význam jeho jednotlivých částí, samotný jazyk XML ovšem nedefinuje způsob, jak se konkrétní údaje zobrazí nebo vytisknou. Můžeme si proto zvlášť vytvořit definici vzhledu jednotlivých elementů, které se říká styl. S jeho pomocí je již zobrazení dokumentu možné, stačí použít aplikaci, která umí číst XML dokumenty a rozumí použitému stylovému jazyku.

Když vznikal jazyk XSL, umožňoval definovat vzhled jednotlivých elementu – způsob jejich zarovnání, velikost a styl písma, barvy atd. Kromě toho jej šlo použít i k automatickému generování obsahu, číslování obrázku, kapitol atd. Jazyk XSL tedy slouží ke dvěma věcem – k transformaci XML dokumentů a k definici vzhledu jejich formátování. Během příprav standardu XSL z něj byla vyřazena část sloužící k transformaci dokumentů, pro kterou se dnes používá název XSLT. Pomocí XSLT lze vytvářet styly definující jakým způsobem se mají XML dokumenty převádět do jiných formátů jako např. HTML, do XML dokumentu s jinou strukturou nebo do obyčejných textových souborů. Druhé části XSL, která slouží k přesnému popisu vzhledu dokumentu, se říká XSL FO (formátovací objekty).[3, 4, 5]



Obrázek 3: Transformace dokumentu s použitím stylu

3.3.2 Šablony

Základem každého stylu jsou šablony. Jejich základní tvar je:

```
<xsl:template match="vzor">
    tělo šablony
</xsl:template>
```

Tělo šablony přesně definuje, jak se části vyhovující výrazu budou zpracovávat. V těle šablony můžeme používat další konstrukce XSLT nebo přímo i elementy z výsledného dokumentu např. HTML tagy. XPath výraz použitý v atributu `match` nemůže být libovolný XPath výraz. Musí to být takový výraz, který používá pouze osy pro přechod na uzel potomka, atribut nebo znaky „//“.

Mezi dva nejpoužívanější příkazy, které se používají uvnitř šablony, patří `value-of` a `apply-templates`. V závislosti na použitém XSLT procesoru může vypadat zpracování šablon například následovně: XSLT procesor na začátku své práce načte do paměti vstupní XML dokument a vytvoří si jeho stromovou reprezentaci. Tento strom je pak postupně procházen od kořene v pořadí, v jakém jsou elementy obsaženy v dokumentu. V okamžiku, kdy je nalezena šablona odpovídající uzlu ve stromu, začne se její obsah zapisovat na výstup. Další potomci uzlu, pro který byla vybrána šablona, už nejsou dál automaticky zpracováváni. Pokud je chceme zpracovat, musíme uvnitř šablony použít instrukci `<xsl:apply-templates>`. Ta udává, že se má daná větev stromu zpracovávat dále a mají se pro její uzly hledat odpovídající šablony.
Pokud chceme v těle šablony použít jen textový obsah daného elementu a jeho podelementu, ale nechceme aplikovat další šablony, používá se instrukce `<xsl:value-of select="výraz">`. Ta vybere pouze obsah textových uzlů, které jsou potomky elementu určeného pomocí výrazu. Pro vyvolávání šablon je tedy důležité pořadí elementu v dokumentu a nikoliv poradí šablon ve stylu.

Výběr uzlů

Pokud uvnitř šablony použijeme `<xsl:apply-templates/>`, začnou se hledat šablony pro všechny potomky aktuálního uzlu. Pokud chceme, aby se hledaly šablony pro jinou část dokumentu, můžeme ji určit použitím atributu `select`. Uvedeme si jednoduchý okomentovaný příklad stylu. Postupně vybereme všechny knihy v knihovně a vypíšeme do tabulky jejich název a rok vydání.

```
1  <!-- určení použitého kódování -->
2  <?xml version="1.0" encoding="utf-8"?>
3
4  <!-- definice stylu -->
5  <xsl:stylesheet xmlns:xsl="http :// www.w3.org/1999/XSL/Transform" version="1.0">
6
7  <!-- definice výstupního formátu HTML -->
8  <xsl:output method="html" encoding="utf-8"/>
9
10 <!-- šablona bude použita na celý dokument -->
11 <xsl:template match="/">
12  <html>
13    <head>
14      <!-- vypíše název knihovny (text v cestě /knihovna/nazev)-->
15      <title>Knihovna <xsl:value-of select="knihova/nazev"/></title>
16    </head>
17    <body>
18      <h1>Knihovna <xsl:value-of select="knihova/nazev"/></h1>
19
20      <table>
21        <!-- vybere šablonu 'kniha' pro zpracování informací o knize -->
22          <xsl:apply-templates select="//kniha"/>
23      </table>
24    </body>
25  </html>
26 </xsl:template>
27
28 <!-- šablona kniha, obsahuje šablony 'nazev' a 'rok' -->
29 <xsl:template match="kniha">
30  <tr>
31    <xsl:apply-templates select="nazev|rok"/>
32  </tr>
33 </xsl:template>
34
35 <!-- šablona 'nazev' -->
36 <xsl:template match="nazev">
37  <td><xsl:apply-templates/></td>
38 </xsl:template>
39
40 <!-- šablona 'rok' -->
41 <xsl:template match="rok">
42  <td><xsl:apply-templates/></td>
43 </xsl:template>
44
45 </xsl:stylesheet>
```

Výpis 3: Příklad aplikace stylu

Režimy zpracování šablon

Pokud potřebujeme některé uzly dokumentu zpracovat opakovaně pokaždé jiným způsobem, můžeme u každé šablony definovat režim pomocí atributu `mode`. Například

můžeme chtít výpis knih zformátovat tak, aby první ve výpisu byla přehledná tabulka s názvem knihy a rokem vydání a pod ní pak však podrobnosti o knize. Většinu uzlů v dokumentu tak musíme zpracovat dvakrát, pokaždé však jiným způsobem.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <xsl:stylesheet xmlns:xsl="http :// www.w3.org/1999/XSL/Transform" version="1.0">
3  <xsl:output method="html" encoding="utf-8"/>
4
5  <!-- šablona bude použita na celý dokument -->
6  <xsl:template match="/">
7    <html>
8      <head>
9        <!-- vypíše název knihovny (text v cestě /knihovna/nazev)-->
10       <title>Knihovna <xsl:value-of select="knihova/nazev"/></title>
11     </head>
12     <body>
13       <h1>Knihovna <xsl:value-of select="knihova/nazev"/></h1>
14
15       <table>
16         <!-- vybere šablonu 'kniha' pro zpracování informací o knize -->
17           <xsl:apply-templates select="//kniha"/>
18       </table>
19
20       <xsl:apply-templates select="//kniha" mode="detail" />
21     </body>
22   </html>
23 </xsl:template>
24
25 <!-- šablona kniha, název a rok jsou shodné s předchozí ukázkou -->
26
27 <!-- šablona kniha s režimem 'detail'-->
28 <xsl:template match="kniha" mode="detail">
29   <tr>
30     <xsl:apply-templates mode="detail"><HR/>
31   </tr>
32 </xsl:template>
33
34 <!-- šablona 'zanr' -->
35 <xsl:template match="zanr" mode="detail">
36   <em><xsl:apply-templates/></em>
37 </xsl:template>
38
39 <!-- šablona 'cena' -->
40 <xsl:template match="cena" mode="detail">
41   <em><xsl:apply-templates/></em>
42 </xsl:template>
43
44 </xsl:stylesheet>
```

Výpis 4: Režimy zpracování šablon

Iterativní zpracování

Uvnitř libovolné šablony můžeme použít instrukci `<xsl:for-each>`. Obsah elementu `<xsl:for-each>` tvoří šablonu, která se zpracovává pro každý uzel zpracovávaný v tomto cyklu. Uzly ke zpracování se vybírají pomocí atributu `select` a výraz by měl vybírat množinu uzlů. Vybrané uzly jsou zpracovány v pořadí, v jakém se vyskytují v dokumentu. Toto pořadí lze změnit pomocí instrukce pro seřazování: `<xsl:sort>`. V následující ukázce se vyberou všechny názvy a roky vydání a vypíšou se do jednotlivých řádků tabulky.

```
<xsl:for-each select="//kniha">
   <tr>
   <td><xsl:value-of select="nazev"/></td>
    <td><xsl:value-of select="rok"/></td>
   </tr>
</xsl:for-each>
```

Podmíněné zpracování

Pomocí `<xsl:if>` můžeme podmíněně vykonat část XSLT kódu. Podmínka se zadává v atributu `test` a musí to být XPath výraz, který vrací logickou hodnotu. Pokud je tento výraz pravdivý, podmíněná část se provede. Zápis podmínky vypadá asi takto:

```
<xsl:if test="podmínka">
  příkazy
</xsl:if>
```

3.4 Procesor Saxon

Balíček SAXON je kolekce nástrojů pro zpracovávání XML dokumentů.[7, 8] Hlavními komponentami tohoto balíčku jsou:

- XSLT 2.0 procesor, který může být použit z příkazové řádky nebo volán z aplikace pomocí API
- XPath 2.0 procesor přístupný aplikacím přes API
- XQuery 1.0 procesor, který může být použit z příkazové řádky nebo volán z aplikace pomocí API
- XML Schema 1.0 procesor, který umožňuje validovat správnost schémat i validovat dokumenty vůči těmto schématům

Saxon je distribuován ve třech verzích balíčků: Saxon-HE, Saxon-PE a Saxon-EE. Každý z těchto balíčků je dostupný jak pro platformu Java, tak i pro .NET. Pouze Saxon-HE je dostupný pod volně šiřitelnou licencí (open-source), ale obsahuje základní funkce pro práci s XSLT a XQuery, což nám bude v tomto projektu stačit.

Pro spuštění Saxonu musíme mít nainstalovanou Java Virtual machine (JVM), doporučuje se mít Java Development kit (JDK) distribuci. Aktuální verze Saxonu 9.2 potřebuje pro korektní chod JDK 1.5 a novější.

4 Analýza a návrh aplikace

4.1 Transformace dokumentu

Abychom správně pochopili podstatu této práce, musíme si nastínit podstatu řešeného problému. XML dokument je navržen pro uchovávání velkého objemu dat textového charakteru a s tímto návrhem je spojena i nutnost v takto vytvořených dokumentech efektivně hledat. Problémem rozsáhlých XML dokumentů je relativně zdlouhavé vyhledávání při dotazech na konkrétní hodnoty uzlů dokumentu. Ve velkých XML dokumentech bývá struktura složitá a uzly mohou mít řadu různých vlastností. Když poté v této složité struktuře vyhledáváme uzly s určitými vlastnostmi, je třeba testovat každý uzel a porovnávat jeho vlastnosti s vlastnostmi hledanými.

Jako jedním z řešení tohoto problému se jeví vhodná optimalizace struktury XML dokumentu do podoby, která umožní rychlejší vyhledávání nad těmito daty. Jednou z možností je vkládání uměle vytvořených klíčů (elementů) do struktury XML dokumentu. Tím se docílí lepšího rozložení stromové struktury souboru a pro vyhledávání nad těmito daty, na které je optimalizovaný XML soubor transformován, se nemusí procházet celý dokument, ale jen jeho část. Tímto lze docílit časové úspory při procházení dokumentem.

Pro pochopení řešení tohoto problému si musíme definovat pojmy, které si dále rozebereme. Pro transformaci dokumentu si musíme zvolit dva typy značkovaných cest. Prvním typem cesty bude *transformační cesta*. Transformační cesta nám bude udávat pozici uzlu v dokumentu, podle jehož vlastnosti, zadané *hodnotovou cestou*, bude tento uzel i s potomky přesunut v rámci optimalizace struktury dokumentu. Uzlu definovaného transformační cestou budeme říkat *transformační uzel*.

Hodnotová cesta nám poté bude udávat uzel, podle jehož obsahu se bude transformační uzel přesouvat do nově vznikající struktury. Uzlu definovaného hodnotovou cestou budeme říkat *hodnotový uzel*. Vždy tedy musí platit, že hodnotový uzel je přímým potomkem transformačního uzlu.

Myšlenka tohoto řešení je jednoduchá. Pro všechny transformační uzly zjistíme obsah jejich hodnotových uzlů. Poté při transformaci dokumentu na úrovni transformačních uzlů vytvoříme uzly nové, jejichž název bude odpovídat unikátním hodnotám získaných v předchozím kroku. Podle hodnoty jednotlivých hodnotových uzlů přesuneme transformační uzly a jejich potomky pod nový uzel, jehož název odpovídá hodnotě hodnotového uzlu. Tento postup opakujeme pro všechny uzly ležící na transformační cestě.

Poté již pro hledání uzlu s určitou vlastností nemusíme procházet celý dokument, ale jen jeho část. Tato metoda je tím účinnější, čím více sousedů mají dané transformační uzly a čím méně různých hodnot mají jejich hodnotové uzly. Také má nezanedbatelný vliv, jak hluboko ve struktuře dokumentu je transformační uzel umístěn. Platí, že čím blíže ke kořeni dokumentu se transformační uzel nachází, tím méně struktury dokumentu musíme prohledat, abychom našli dotazovaný uzel.

Příklad

Pro lepší pochopení si uvedeme následující příklad transformace. Mějme strukturu XML dokumentu uvedenou ve následujícím výpisu. Vlevo je uvedena struktura XML před transformací a vpravo po transformaci. Rozhodli jsme se, že budeme chtít `knihy` v knihovně seřadit podle jejich `autora`. Požadovanými vstupy pro transformaci tedy budou:

- transformační cesta: /knihovna/kniha
- hodnotová cesta: /knihovna/kniha/autor

```
<knihovna>
  <kniha>
    <nazev>Kniha1</nazev>
    <autor>Autor1</autor>
    <rok>Rok1</rok>
  </kniha>
  <kniha>
    <nazev>Kniha2</nazev>
    <autor>Autor2</autor>
    <rok>Rok2</rok>
  </kniha>
  <kniha>
    <nazev>Kniha3</nazev>
    <autor>Autor1</autor>
    <rok>Rok3</rok>
  </kniha>
</knihovna>
```

```
<knihovna>
  <Autor1>
    <kniha>
      <nazev>Kniha1</nazev>
      <autor>Autor1</autor>
      <rok>Rok1</rok>
    </kniha>
    <kniha>
      <nazev>Kniha3</nazev>
      <autor>Autor1</autor>
      <rok>Rok3</rok>
    </kniha>
  </Autor1>
  <Autor2>
    <kniha>
      <nazev>Kniha2</nazev>
      <autor>Autor2</autor>
      <rok>Rok2</rok>
    </kniha>
  </Autor2>
</knihovna>
```

Výpis 5: Příklad transformace

Z výpisu je zřejmé, že elementy `kniha` se přesunuly pod nově vytvořené elementy `Autor1` a `Autor2` v závislosti na tom, jakou hodnotu nabýval jejich hodnotový element. Toto je základní princip námi požadované transformace. Při hledání knih od autora „Autor1“ se poté nemusí prohledávat celý XML soubor, ale stačí prohledat jen knihy pod elementem „Autor1“. V tomto jednoduchém příkladu se rozdíl v hledání zdá nepodstatný, ale v dokumentu s mnoha tisíci elementy a složitou strukturou je již přínos velmi výrazný.

4.2 Problémy transformace

Při transformaci dokumentu dochází k několika problémům, které z tohoto přístupu vyplývají. Je třeba tyto problémy určit a zamyslet se nad možnostmi jejich řešení.

Více hodnotových cest

V XML dokumentu může nastat situace, kdy k jedné transformační cestě existuje více cest hodnotových. Pokud tyto hodnotové cesty obsahují stejné řetězce, transformaci to neovlivní. Pokud ale každá z hodnotových cest obsahuje jiný řetězec znaků, nastává problém, pod jaký element tento transformační uzel přesunout. Variant řešení je několik:

1. vzít hodnotu na první hodnotové cestě
2. poskládat řetězec z hodnot více hodnotových uzlů

Při návrhu programu byla použita druhá varianta, kdy je výsledný řetězec poskládán z hodnot několika hodnotových uzlů. Před samotným skládáním výsledného řetězce jsou hodnoty seřazeny podle abecedy a teprve pak spojeny. Je tak zajištěna částečná ochrana proti výskytu hodnot na hodnotových uzlech v přeházeném pořadí.

Chybějící hodnotová cesta

V XML dokumentu může také nastat situace, kdy u jedné nebo více transformačních cest neexistují cesty hodnotové. V tomto případě se transformační uzel nepřesunuje pod žádnou novou strukturu, ale ponechá se na svém původním umístění.

Nepřípustné znaky

Na hodnotové cestě se může vyskytovat i řetězec složený z více slov. Tímto vzniká mezera mezi slovy, která je jedním z nepřípustný znaků při vytváření názvu nových elementů. Programově toto řešení nevadí, ale dokument je poté ne správně strukturovaný. Možnost, jak tuto chybu napravit, je programové nahrazení mezery mezi slovy jiným zástupným znakem. V programu je implementace nahrazení mezery znakem „_“ (dolní podtržítko). Poté je například řetězec „Karel Čapek“ upraven na řetězec „Karel_Čapek“.

4.3 Analýza požadavků

Cílem této implementace je vytvořit nástroj pro transformaci XML dokumentů na základě uživatelem zvolených parametrů. Práce s programem by měla být rozdělena do tří fází. V **první fázi** by měl výsledný program dokázat analyzovat vstupní XML dokument a podle vstupních parametrů analyzovat jeho obsah. Výsledky této analýzy by měly být uloženy v databázi pro další zpracování. Výsledkem analýzy bude zjištění struktury dat v dokumentu. Tyto statistiky budou poté prezentovány uživateli.
Ve **druhé fázi** budou uživateli prezentovány statistiky získané v první fázi a uživateli bude umožněno přehlednou formou vybrat cesty pro transformaci XML dokumentu.

Uživatel bude mít možnost analytického zjištění, kolik nových elementů bude vytvořeno a kolik stromových struktur pod těmito novými uzly bude transformací ovlivněno. Na základě těchto statistik se bude moci rozhodnout, zdali je zvolená hodnotová cesta pro transformaci výhodná či nikoliv.
Ve **třetí fázi** bude následně provedena samotná transformace XML dokumentu podle zvolených cest. Vstupní XML dokument bude načítán a pro zadané cesty modifikován do výstupního XML dokumentu. Je třeba podotknout, že nebude modifikovaná celá struktura XML dokumentu, ale pouze jeho část, která bude odpovídat zvolené transformační cestě. Části, které nebudou modifikovány, budou kopírovány ze vstupního dokumentu do výstupního v nezměněné podobě.

4.4 Programová specifikace

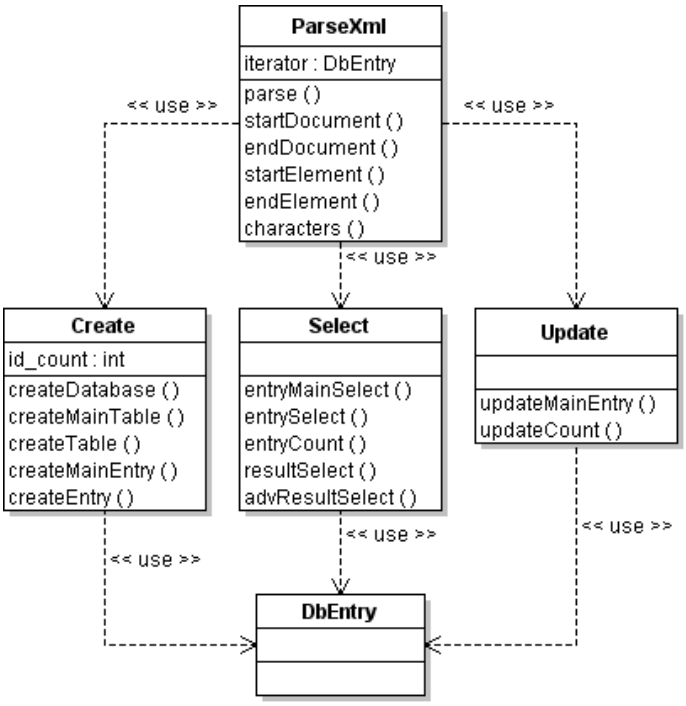
Pro splnění programové části této diplomové práce jsem zvolil implementační prostředí **Eclipse** [11], založené na platformě **Java** [10] společnosti Sun Microsystems. V tomto objektově orientovaném prostředí jsem implementoval všechny funkce programu včetně grafického uživatelského rozhraní. Toto uživatelské rozhraní bylo implementováno pomocí Eclipse pluginu **Jigloo**.[12] Pro vygenerování struktury tříd konfiguračního nastavení programu byl použit plugin JAXB. [13]
Použitá verze platformy Java byla JDK 1.6.0 update 16 a verze vývojového prostředí Eclipse 3.5.1 . Jako řešení databázové struktury byla zvolena aplikace **MySQL** verze 5.1.

4.5 Struktura programu

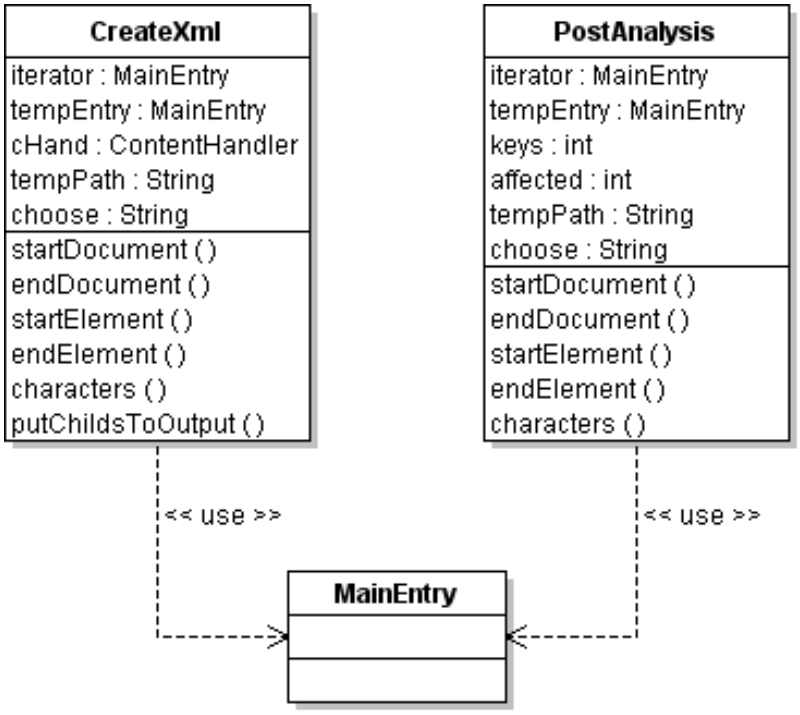
Program se skládá z jednotlivých balíků, které jsou rozděleny podle své funkce. Jednotlivé balíky obsahují vlastní třídy s logikou programu a jejich struktura je následující:

- `actions` - obsluha akcí uživatelského rozhraní
- `config` - práce s konfiguračními proměnnými
- `main` - obsahuje uživatelský interface
- `object` - třídy pomocných objektů pro práci programu
- `parsing` - třídy pro práci s XML dokumentem
- `sql` - třídy pro práci s databází
- `states` - obsluha stavů programu

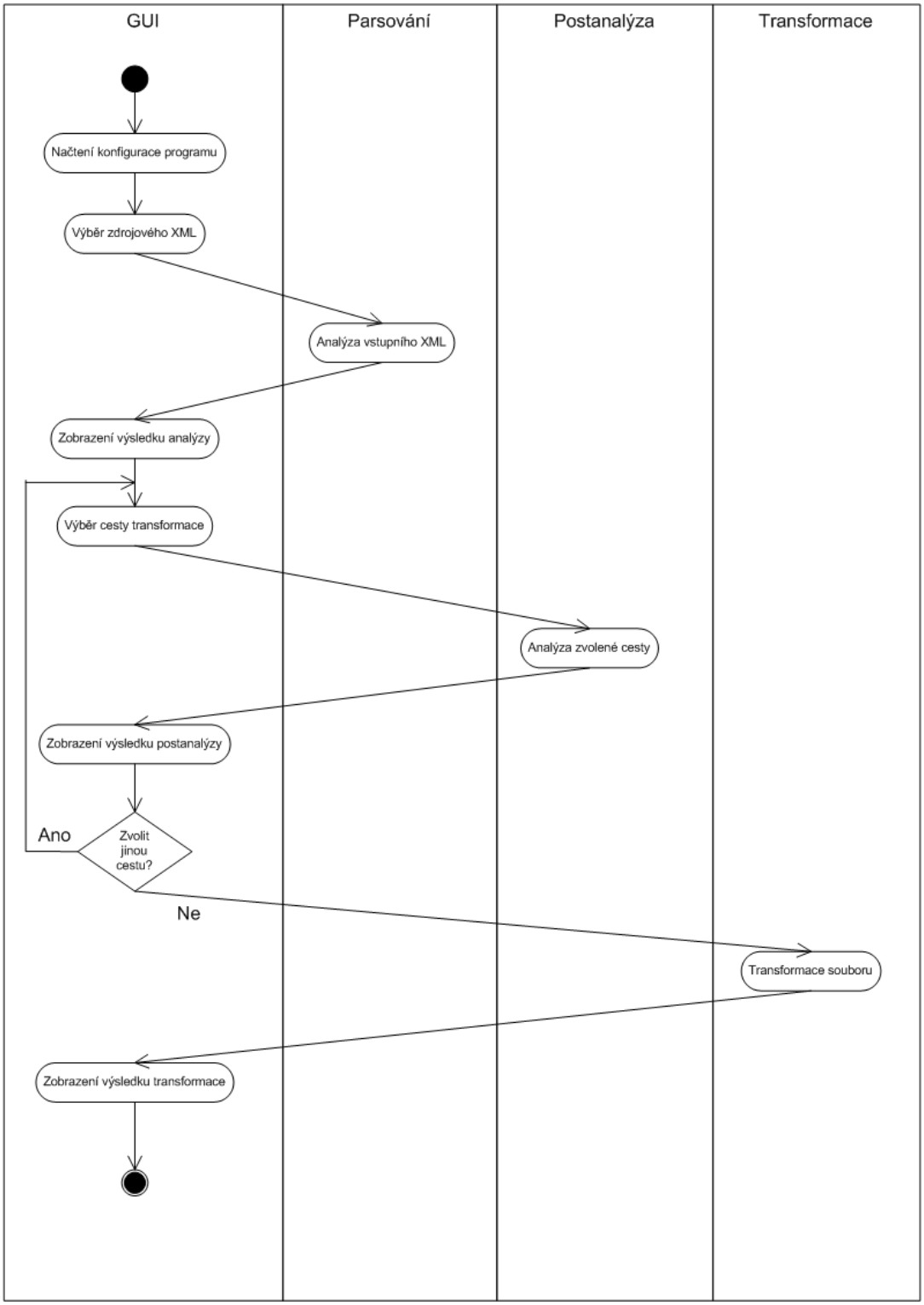
Na následujících třídních diagramech můžete vidět strukturu jednotlivých částí aplikace.



Obrázek 4: Třídní diagram analýzy dat



Obrázek 5: Třídní diagram transformace



Obrázek 6: Aktivitní diagram transformace

5 Analýza vstupních dat

Analýza zdrojového XML dokumentu nám slouží k vytvoření představy o struktuře dat v tomto dokumentu a jejich četnosti. Na základě těchto statistik jsme poté schopni nabídnout nejvhodnější hodnotové cesty pro generování umělých klíčů a vylepšit tak práci s transformovaným dokumentem podle našich požadavků. XML dokument je programem procházen a každý element je příslušně zpracován a zapsán do tabulky. Při procházení XML dokumentem si ke každému elementu na cestě pamatujeme několik atributů, které nám poté slouží pro vybírání nejvhodnějších řetězců. Tyto parametry jsou:

- cesta k jednotlivým záznamům ve tvaru `/root/element1/element2` apod.
- počet výskytů stejných cest k jednotlivým záznamům
- maximální počet záznamů se stejnou hodnotou na stejné cestě
- průměrný počet záznamů se stejnou hodnotou na stejné cestě (kapitola 5.5)
- počet záznamů na stejné cestě s různou hodnotou

K jednotlivým záznamům je ukládána do databáze i jejich hodnota. Z kapacitních důvodů může být počet zapamatovaných různých hodnot u každé z cest omezen parametrem v konfiguračním souboru (viz uživatelská příručka).

Jako výsledek této analýzy jsou uživateli poskytnuta data s vybranými záznamy jako nejvhodnějšími kandidáty na modifikaci výsledného XML dokumentu. Uživatel si poté sám rozhodne, která varianta mu vyhovuje nejvíce. Analýza má tedy pouze zprostředkovat uživateli obsah a strukturu četnosti zdrojového XML dokumentu do přívětivé a pochopitelné formy.

5.1 Načtení konfigurace programu

Při startu analytické části programu se nejprve načte seznam hodnot z konfiguračního XML souboru `config.xml`. Tyto hodnoty jsou poté zapsány do příslušných tříd balíku `config`. Z tohoto umístění poté program načítá názvy proměnných a nastavení konfigurace. Názvy proměnných se shodují s názvy v konfiguračním XML souboru.

5.2 Tvorba databázových tabulek

Pro účely analýzy je pro každou instanci programu vytvořena nová databáze a v ní dvě tabulky. Třída `Create` v balíku `sql` obsahuje metody pro tvorbu databáze a tabulek. V metodě `createDatabase()` vytvoříme nejprve samotnou databázi a pak metodami `createMainTable()` a `createTable()` vytvoříme samotné tabulky. Schéma návrhů tabulek je ukázáno v datovém modelu níže.[14]

Lineární zápis:

main (id, name, count, max, avg, diff)
attributes (id, value)

Datový slovník:

položka	dat. typ	délka	prim. klíč	NULL	index	význam
id	integer	20	Ano	Ne	Ne	identifikátor záznamu
name	varchar	255	Ne	Ne	Ano	cesta v dokumentu
count	integer	20	Ne	Ne	Ne	počet výskytu cesty
max	integer	20	Ne	Ne	Ne	maximální počet sousedů
average	integer	20	Ne	Ne	Ne	průměrný počet sousedů
different	integer	20	Ne	Ne	Ne	počet různých hodnot

Tabulka 1: Datový slovník tabulky main

položka	dat. typ	délka	prim. klíč	NULL	index	význam
id	integer	20	Ano	Ne	Ne	identifikátor záznamu
value	varchar	765	Ano	Ne	Ne	hodnota záznamu

Tabulka 2: Datový slovník tabulky attributes

První tabulka `main` obsahuje jednotlivé cesty v XML dokumentu a jejich parametry. Za zmínku stojí atribut `different`, který uchovává informaci o tom, kolik existuje různých hodnot záznamu na dané cestě. Tato hodnota je klíčovou při volbě hodnotové cesty při transformaci XML dokumentu. Limitní hodnota různých záznamů na dané cestě je konfigurovatelná a pokud je překročena, neukládají se do databáze nové unikátní hodnoty (položky atributu `value`). Limit tím pádem nastavuje hranici, do jaké míry nás zajímá četnost různých hodnot analyzovaných cest.

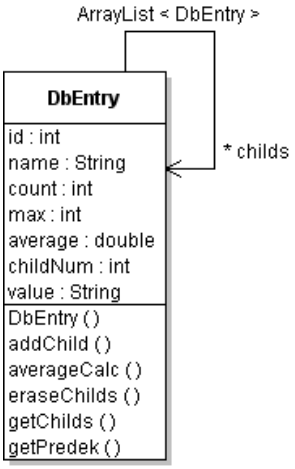
Druhá tabulka `attributes` obsahuje ID záznamu z první tabulky a hodnotu záznamu. Jako hodnoty záznamu `value` jsou zapisovány pouze unikátní hodnoty. To znamená, že pokud již cesta obsahuje porovnávanou hodnotu, hodnota se již do této tabulky nezapíše. Pokud cesta neobsahuje žádný záznam, tak pro ně nebudou existovat ani záznamy v tabulce atributů.

Pro úspěšné provádění veškerých operací nad databází je potřeba vytvořit propojení programů s databází. Toto je zajištěno instancí třídy `Connect`. Tato třída obsahuje metodu `getConnection()` spolupracující s balíkem mysql connector a vytvoří spojení `conn`, přes jehož rozhraní můžeme provádět příkazy v SQL formátu.

5.3 Objekt záznamu DbEntry

Jako vhodnou strukturu pro uchování informací o každém elementu jsem si vytvořil datový objekt DbEntry, který obsahuje všechny potřebné informace o daném elementu. Výhodou tohoto řešení je, že ke všem informacím lze přistupovat jednotně pomocí unifikovaných metod, stejných pro každý element. Je zde tedy naplno využit objektově orientovaný přístup k datům. Toto řešení má své nevýhody v teoreticky větší náročnosti na paměť a bude předmětem testů, jak moc se tato náročnost projeví na běhu programu. Jak již bylo zmíněno, pro každý načtený záznam na cestě v XML dokumentu je vytvořena instance objektu DbEntry. To znamená, že pro každý načtený element z XML dokumentu je vytvořen záznam DbEntry. Každý tento objekt obsahuje tyto vlastnosti:

- `int id` - identifikační číslo záznamu v tabulce (primární klíč)
- `String name` - cesta v dokumentu
- `int count` - celkový počet výskytu dané cesty v dokumentu
- `int max` - maximální počet sousedů dané cesty
- `double average` - průměrný počet sousedů dané cesty
- `int childNum` - počet potomků na dané cestě
- `String value` - hodnota elementu na cestě
- `DbEntry parent` - odkaz na rodičovský element
- `Arraylist<DbEntry> childs` - pole se seznamem potomků



Obrázek 7: Struktura objektu DbEntry

5.4 Parsování

Parsování XML dokumentu je obsaženo ve třídě `ParseXml`. Parsování je implementováno za pomocí API SAX. Z tohoto API využíváme třídu `DefaultHandler`, ve které přepisujeme metody zpracovávající události, konkrétně:

- `startDocument()`
- `endDocument()`
- `startElement()`
- `endElement()`
- `characters()`

Metody jsou při procházení XML dokumentu volány automaticky. Metoda `startDocument()` se provede při načtení počátečního tagu XML dokumentu, `endDocument()` při načtení ukončovacího tagu XML dokumentu. Metoda `startElement()` se volá při každém načtení nového elementu a `endElement()` při opuštění elementu. Metoda `characters()` obsahuje logiku načítání řetězce znaků mezi počátečním a koncovým tagem elementu. Do těchto metod si může programátor vložit svůj kód a bude mít jistotu, že bude proveden ve správný čas.

Při vytvoření nové instance třídy `Parse` je vytvořena instance třídy `XMLReader()` a je jí předána instance třídy `ParseXml` jako handler pro zpracování událostí. V analytické části se při načtení počátečního a ukončovacího tagu XML dokumentu vypíše pouze oznámení uživateli o začátku či konci analytické části. Žádná další funkcionalita se v metodách `startDocument()` a `endDocument()` neřeší. Zajímavější metody jsou pro načítání začátku a konce elementu.
Při inicializaci je vytvořen základní objekt `DbEntry` (viz kapitola 5.3) s názvem `iterator`, který slouží jako ukazatel na aktuální uzel námi vytvářené stromové struktury. Před načtením prvního elementu se takto stává kořenovým uzlem. Tento kořenový uzel má svůj seznam potomků, kde se postupně ukládají načítané elementy (také ve formě objektu `DbEntry`) z XML dokumentu.

startElement()

Při načtení nového elementu se zavolá metoda `addChild()` ukazatele aktuálního uzlu, pro tento nový element vytvoří objekt `DbEntry` a přidá se do seznamu potomků `childs` ukazatele aktuálního uzlu. Následně se ukazatel přesune na právě vytvořený objekt.

endElement()

Když parser narazí na koncový tag elementu, zavolá metodu `endElement()`. Pro tento element, který je nyní označen jako `iterator`, si vybereme seznam všech jeho potomků

`childs` metodou `getChilds()`. Poté procházíme jednotlivé potomky a u každého zjišťujeme, zdali se název potomka (jeho cesta v XML dokumentu) již nevyskytuje v databázi. Realizace probíhá tak, že v hlavní tabulce `main` hledáme metodou `entryMainSelect()` všechny položky se shodnou cestou jako má aktuální potomek. Podle výsledku hledání mohou být realizovány následující 2 možnosti.

Pokud název nebyl v tabulce nalezen, je instance `DbEntry` potomka předána metodě `createMainEntry()`, která vytvoří nový záznam podle obsahu potomka do hlavní tabulky `main`. Každému novému záznamu je přiřazeno unikátní `id` číslo, aby byla zaručena jedinečnost záznamu. Toto `id` číslo je provázáno s tabulkou atributů `attributes`, kde jsou uloženy hodnoty záznamů ve tvaru (id, hodnota). Pokud se jedná o koncový element tak nutné ještě vytvořit záznam v tabulce atributů, který vytvoříme pomocí metody `createEntry()`.
Druhá možnost nastává, pokud již byl záznam v tabulce nalezen. Zjistíme, zdali záznam má sousedy se stejným názvem jako on. Pokud ano, musíme provést přepočet průměrného výskytu záznamu (viz kapitola 5.5) a aktualizovat jej v hlavní tabulce. Dále pokud je záznam listem (koncovým elementem), musíme přidat jeho hodnotu do tabulky atributů k příslušné cestě pokud již tento záznam neexistuje. To uděláme tak, že si vytáhneme `id` číslo záznamu z hlavní tabulky a s tímto číslem jej uložíme do tabulky atributů jako dvojici (id, hodnota). Počet záznamu hodnot ke každé z cest v dokumentu je omezen a můžeme jej nastavovat v konfiguračním XML souboru pod položkou `different`.

Nakonec posuneme ukazatel `iterator` na element o úroveň výše. Tímto zajistíme zpracovávání následujícího elementu.

characters()

Tato metoda zajišťuje načítání obsahu nacházejícího se mezi počáteční a ukončovací značkou elementu. Je zde také implementována logika zápisu do tabulky atributů (viz kapitola 5.2) a možnost omezení délky řetězce ukládaného do databáze.

5.5 Výpočet průměrného výskytu záznamu

Dobrým ukazatelem a námi uchovávaným atributem je i průměrný počet výskytu záznamů na dané cestě. Zatímco maximální počet záznamů a počet různých hodnot záznamu lze zjistit jednoduchým databázovým dotazem, průměrný počet výskytů záznamů si musíme pouze vypočítat. Na následujícím obrázku 8 můžete pro názornost vidět ukázkovou stromovou strukturu dokumentu a výpočet hodnoty.

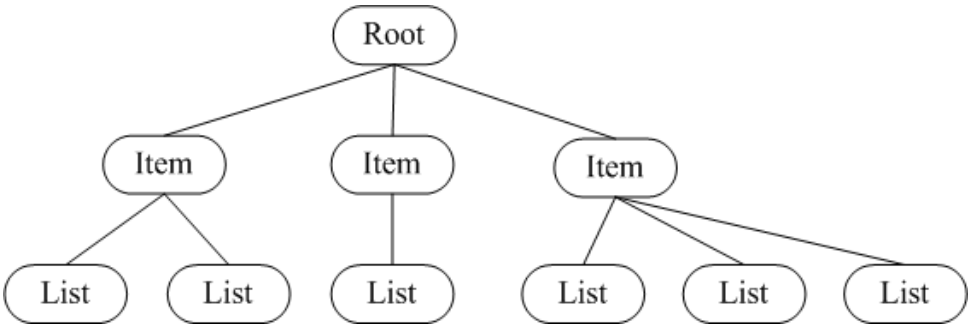
Výpočet se provede v těchto krocích:

1. spočítáme celkový počet rodičů jako: (počet výskytů / průměrný počet záznamů)
2. přičteme aktuální počet sousedských uzlů k počtům výskytů záznamu

3. pokud je nyní aktuální počet sousedů větší než maximální počet sousedů, aktualizujeme i maximální počet záznamů
4. vypočteme nový průměrný počet záznamů jako: (průměr = počet výskytů / (počet rodičů + 1)

Tímto výpočtem aktualizujeme při každém končícím elementu záznamy v tabulce. Pro třetí element item v dokumentu bude tedy daný výpočet vypadat následovně:

1. počet rodičů: 3 / 1.5 = 2
2. počet výskytů záznamu: 3 + 3 = 6
3. aktualizujeme maximální počet záznamů max = 3
4. průměrný počet záznamů nyní je: 6 / 3 = 2



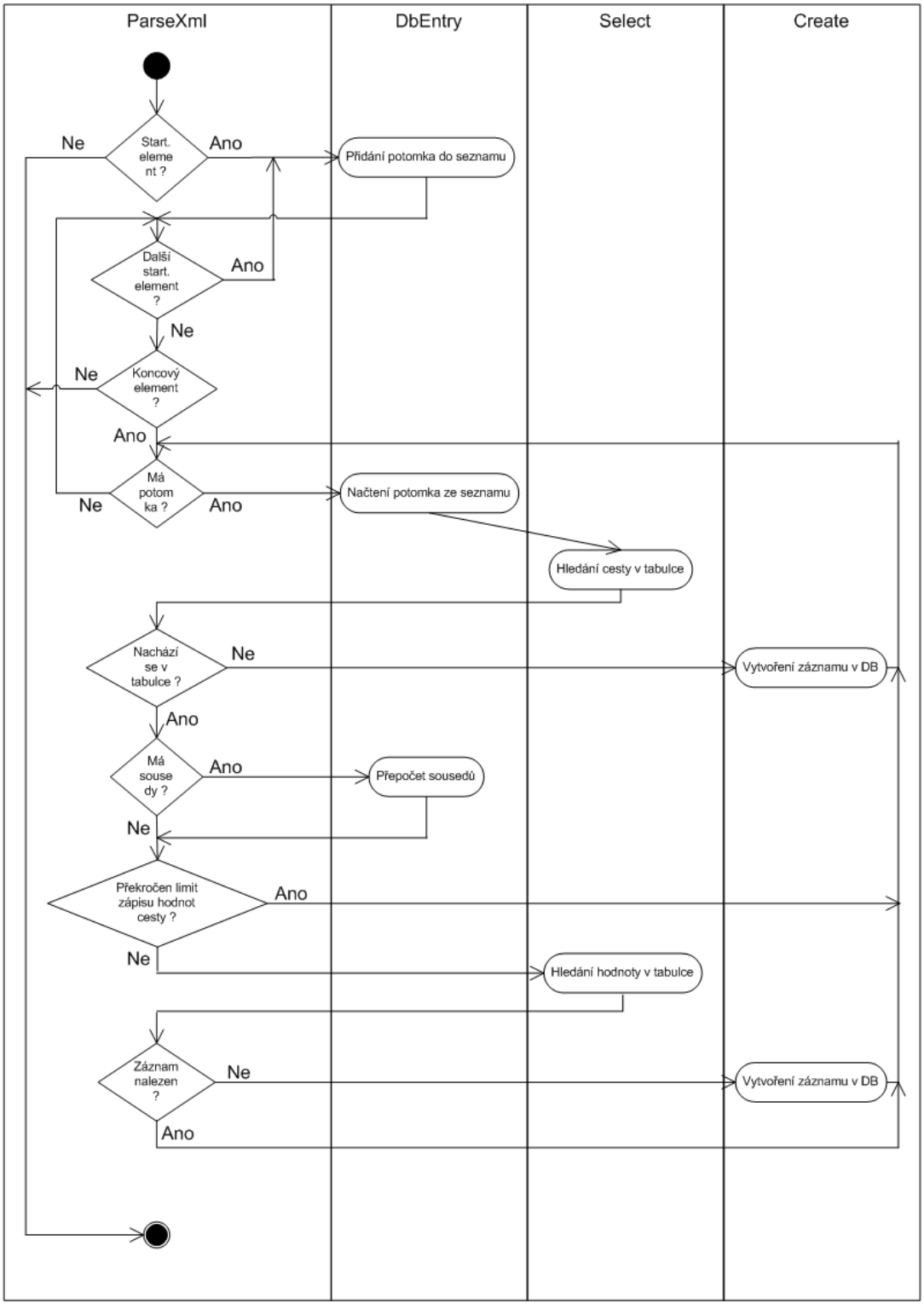
Obrázek 8: Model XML dokumentu pro výpočet průměrného výskytu záznamu

5.6 Analýza paměťové složitosti

Při parsování XML dokumentu je využíván `iterator` obsahující stromovou strukturu načítaného souboru. Tato struktura je poměrně paměťově náročná, tudíž při zpracovávání události ukončení elementu dochází k vymazání všech potomků v seznamu `childs` zpracovávaného elementu. Díky tomuto řešení se odstraní již nepotřebné informace a uvolní se tím místo v paměti.

Nejhorší možný scénář pro navržený program je takový, když má element velmi mnoho přímých potomků, které musí program uchovávat v paměti. Při velmi vysokém počtu těchto přímých potomků může nastat situace, kdy dojde k velikému vytížení paměti, program již nemá kde uchovávat data a aplikace zhavaruje. Naštěstí v běžných XML dokumentech tento jev prakticky nenastává, a proto je program po paměťové stránce dostatečně dimenzován i pro práci s rozsáhlými XML dokumenty.

Paměťovou složitost programu lze zapsat jako: $h \cdot m$, kde h značí maximální výšku stromu dokumentu a m značí maximální počet sousedů. Tuto paměťovou složitost lze také přepsat jako $O(h \cdot m)$.



Obrázek 9: Aktivitní diagram procesu parsování dokumentu

6 Prezentace statistik uživateli

Poté, co proběhne analytická část programu, jsou uživateli předloženy výsledky analýzy. Jsou mu nabídnuty nejvhodnější značkované cesty pro transformaci XML dokumentu. Cesty jsou zobrazeny ve dvou krocích. V prvním kroku jsou zobrazeny transformační cesty a jsou seřazeny podle průměrného počtu sousedů od největšího po nejmenší. Ve druhém kroku si uživatel volí hodnotovou cestu v závislosti na zvolené transformační cestě. Hodnotové cesty jsou zde seřazeny podle počtu různých hodnot, kterých nabývají, a to od nejmenších po největší. Tímto lze preferovat statisticky nejvhodnější transformační a hodnotové cesty pro transformaci.

Uživatel si může, ale také nemusí, z takto doporučených cest vybrat. V případě, že požaduje transformaci dokumentu podle jiné cesty, je mu umožněno tuto cestu vložit a transformovat se bude podle ní. Cesta musí být ovšem zapsána v souladu se syntaxí jazyka XPath pro zápis cest v XML dokumentu (tj. ve tvaru `/root/cesta1/cesta2` apod.).
Poté, co uživatel zvolí jednu z těchto možností, spustí se proces ověřování účinnosti dané hodnotové cesty vůči zdrojovému XML dokumentu - post analýza. Pokud výsledek post analýzy uživateli vyhovuje, dokument se transformuje podle zvolené hodnotové cesty, program informuje uživatele o úspěšné transformaci a vypíše statistiky celého procesu.

6.1 Post analýza

Post analýza je proces, kdy testujeme uživatelem zvolenou hodnotovou cestu na účinnost výsledné transformace dokumentu. Pokud bychom zvolili ne příliš vhodnou cestu, mohlo by se stát, že výsledný XML dokument bude nepřehlednější a vyvážená struktura původního dokumentu bude narušena. Tím pádem obsah dokumentu bude hůře strukturován a hledání v takovémto dokumentu může být náročnější. Smyslem této práce však je transformovat zdrojový XML dokument do zcela opačné podoby - lépe strukturované a vhodnější pro vyhledávání. Proto je tato analýza důležitým prvkem při procesu transformace XML dokumentu.

Rozhodnutí o vhodnosti této transformace závisí na několika faktorech:

- počet uměle (nově) vytvořených elementů v dokumentu
- počet touto transformací ovlivněných struktur (potomků nově vytvořených elementů)
- poměr mezi dvěma předcházejícími hodnotami

Pokud je vypočteno, že transformací XML dokumentu podle zadané hodnotové cesty nedostaneme vhodné uspořádání dokumentu, uživatel je o této skutečnosti informován a je mu nabídnuto zvolení jiné cesty. Pokud uživatel nebude chtít změnit cestu pro transformaci a ponechá původní, bude výsledný dokument transformován podle zadané cesty.

7 Transformace dokumentu

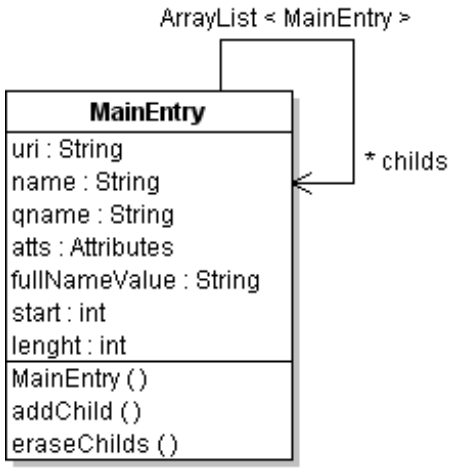
Výstupem transformace XML dokumentu bude XML dokument upravený podle uživatelem zadané hodnotové cesty dokumentu pro transformaci. V prezentační fázi (viz kapitola 6) si uživatel zvolí cestu v dokumentu, podle niž se bude vstupní dokument transformovat. Transformovaný dokument bude mít poté vhodnější strukturu pro vyhledávání podle požadovaných parametrů. Účinnosti jednotlivých transformací budou poté porovnávány se zdrojovým XML dokumentem.

Program bude procházet zdrojový XML dokument a porovnávat aktuální umístění v tomto dokumentu s hodnotovou cestou vybranou uživatelem. Dokud nebude nalezena shoda cesty v dokumentu s vybranou cestou, program bude zapisovat data načtená ze zdrojového dokumentu přímo do výstupního dokumentu. Až bude nalezena shoda aktuálního umístění v dokumentu a vybrané cesty, program analyzuje všechny potomky aktuálního uzlu a podle obsahu těchto potomků bude vhodně upravovat strukturu cílového XML dokumentu. Takto bude program procházet celý zdrojový XML dokument.

7.1 Objekt záznamu MainEntry

Jako v případě datového objektu DbEntry (kapitola 5.3) jsem si vytvořil datový objekt pojmenovaný MainEntry, který obsahuje všechny potřebné informace o daném elementu. Výhodou tohoto řešení je, že ke všem informacím lze přistupovat jednotně pomocí unifikovaných metod, stejných pro každý element. Objekty `MainEntry` se používají při transformaci XML dokumentu pro uchovávání elementů, jejich vlastností a vztahů. Pokud je nalezena shoda cesty v dokumentu s hledanou cestou, tak pro účel analýzy obsahu potomků je pro každý načtený element z XML dokumentu vytvořen záznam `MainEntry`. Tento každý objekt obsahuje tyto vlastnosti:

- `String uri` - cesta v XML dokumentu
- `String name` - název elementu
- `String qname` - název elementu včetně uri
- `Attributes atts` - atributy elementu
- `String fullNameValue` - cesta k elementu
- `MainEntry parent` - předek elementu
- `Char ch[]` - pole s hodnotou elementu
- `int start` - ukazatel na začátek hodnoty elementu v poli ch[]
- `int length` - počet znaků v poli ch[]
- `Arraylist<MainEntry> childs` - pole se seznamem potomků



Obrázek 10: Struktura objektu MainEntry

7.2 Transformace

Transformace XML dokumentu je obsažena ve třídě CreateXml. Transformace je stejně jako parsování implementována za pomocí API SAX. Z tohoto API využíváme třídu DefaultHandler, ve které přepisujeme metody zpracovávající události, konkrétně:

- `startDocument()`
- `endDocument()`
- `startElement()`
- `endElement()`
- `characters()`

Metody jsou při procházení XML dokumentu volány automaticky. Metoda `startDocument()` se provede při načtení počátečního tagu XML dokumentu, `endDocument()` při načtení ukončovacího tagu XML dokumentu. Metoda `startElement()` se volá při každém načtení nového elementu a metoda `endElement()` při opuštění elementu. Metoda `characters()` obsahuje logiku načítání řetězce znaků mezi počátečním a koncovým tagem elementu. Do těchto pěti metod si může programátor vložit svůj kód a bude mít jistotu, že bude proveden ve správný čas.

Při vytvoření nové instance třídy `CreateXml` je vytvořena instance třídy `XMLReader()` a je jí předána instance třídy `CreateXml` jako handler pro zpracování událostí. Také je vytvořen `FileOutputStream`, pomocí kterého budeme zapisovat řetězce znaků na výstup. Stejně jako v analytické části se i při transformaci XML dokumentu při načtení počátečního a ukončovacího tagu XML dokumentu vypíše pouze oznámení uživateli o začátku či konci transformace. Žádná další funkcionalita se v metodách `startDocument()` a

`endDocument()` neřeší.
Při inicializaci je také vytvořen základní objekt `MainEntry` s názvem `iterator`, který slouží jako ukazatel na aktuální uzel námi vytvářené stromové struktury. Před načtením prvního elementu se takto stává kořenovým uzlem. Vytváří se je ještě jeden prázdný objekt `MainEntry` s názvem `tempEntry` (viz strana 38), do kterého se při shodě hledané cesty a aktuálního umístění v dokumentu ukládají načítané struktury elementů. Tento objekt je poté modifikován z hlediska jeho struktury a zapisován na výstup podle obsahu jeho potomků.

startElement()

Při každém načtení počátečního tagu nového elementu pomocí `iteratoru` přesuneme ukazatel na aktuálně načtený element. Pro tento element vytvoříme také nový objekt `MainEntry`. Poté pokaždé testujeme, zdali již jsme narazili na hledaný řetězec a ukládáme tedy načítané elementy do `tempEntry`, nebo jsme na něj ještě nenarazili.
Pokud jsme na řetězec ještě nenarazili, mohou nastat 3 varianty:

1. aktuální cesta v dokumentu neodpovídá cestě elementu pro modifikaci
2. aktuální cesta v dokumentu odpovídá části cesty elementu pro modifikaci
3. aktuální cesta v dokumentu odpovídá cestě elementu pro modifikaci

V prvních dvou variantách pokračujeme načítáním dalšího elementu ze zdrojového dokumentu. Při třetí variantě předáme aktuální uzel jako potomka objektu `tempEntry` a pokračujeme s načítáním dalších elementů.

Pokud jsme již tedy v třetí variantě na řetězec narazili, tak při načtení nového elementu pouze přidáme aktuální uzel pomocí metody `addChild()` do seznamu potomků `childs` struktury `tempEntry`. Tímto zajistíme načtení všech elementů na hledané transformační cestě pro pozdější zpracování.

endElement()

Při každém načtení koncové značky elementu testujeme, zdali jsme v úseku dokumentu, který se bude transformovat nebo ne. Pokud se aktuální pozice elementu v dokumentu neshoduje s porovnávaným řetězcem pro transformaci ani není potomkem tohoto řetězce, předávají se načítané znaky přímo na výstupní stream a zapisují se do souboru.

Pokud se ovšem nacházíme v úseku dokumentu pro transformaci, je situace složitější. Musíme zajistit, abychom načetli koncovou značku řetězce, pro něž transformaci provádíme. Tím zajistíme načtení celého úseku dokumentu pro zpracování. Dokud načítáme koncové značky jiných elementů, přesunujeme ukazatel aktuální pozice na jejich předky. Tímto se posunujeme v hierarchii načtených uzlů směrem ke kořenovému uzlu vytvořené struktury.

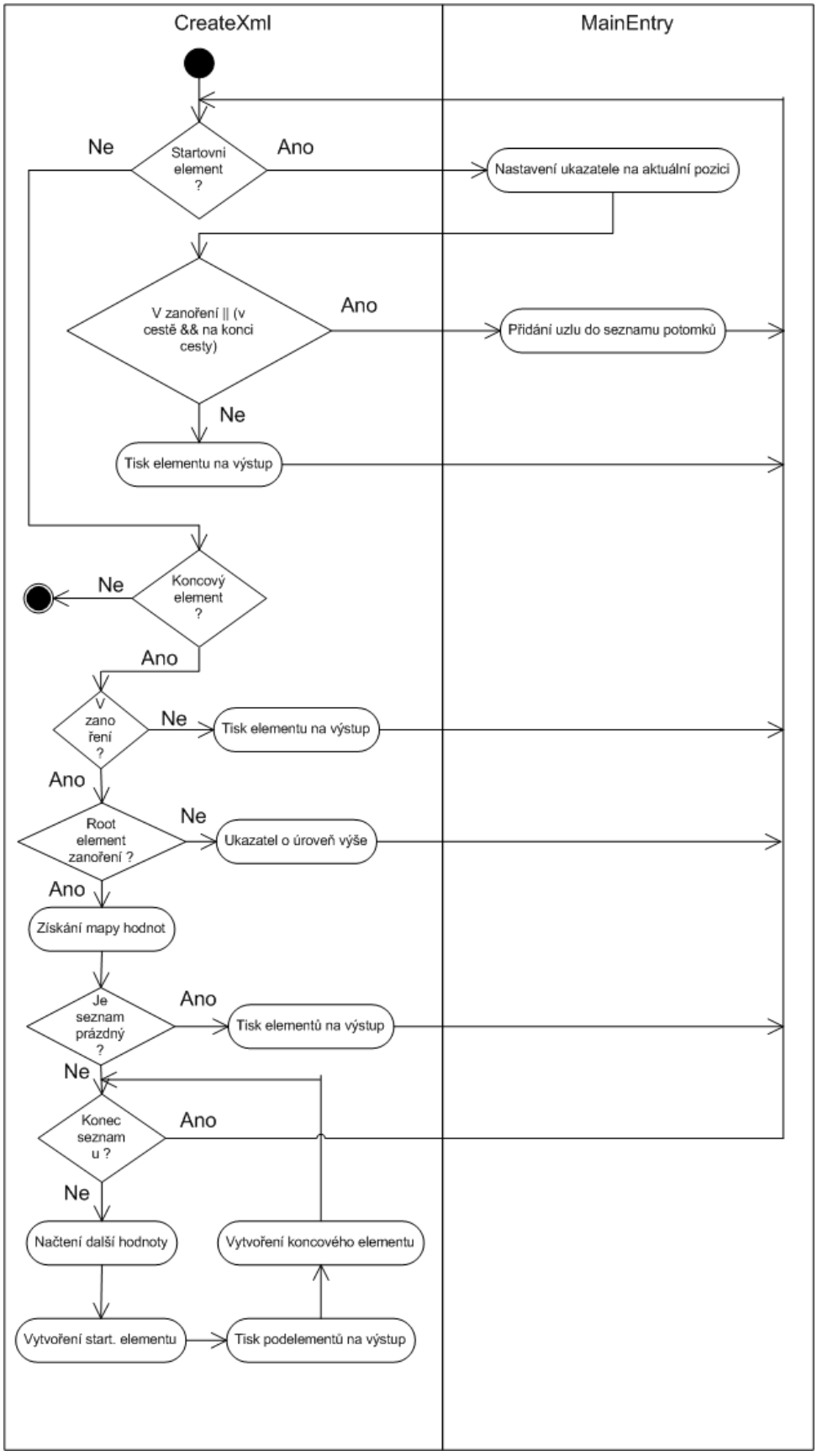
Až načteme koncovou značku řetězce, budeme procházet seznam potomků `childs` tohoto nejvyššího uzlu (`tempEntry`) a pro každého potomka zjistíme hodnotu obsaženou v umístění odpovídající hledané hodnotové cestě. Pokud potomek obsahuje více umístění odpovídající hledané cestě (a tím pádem může, ale nemusí obsahovat i více různých hodnot), jsou tyto hodnoty abecedně setřízeny a spojeny do jednoho řetězce znaků `finalValue`.
Pokud byla nalezena nějaká hodnota, tudíž řetězec `finalValue` je neprázdný, provede se hledání dané hodnoty v hashmapě. Tato struktura slouží k zápisu a hledání hodnot podle daného klíče. V našem případě je klíč hodnota řetězce `finalValue` a každá položka hashmapy obsahuje `ArrayList` se seznamem objektů. Pokud v hashmapě nebyla k danému klíči nalezena žádná položka, je vytvořen nový `ArrayList` a do něj jsou vloženy všichni potomci aktuálního uzlu . Poté je list uložen do hashmapy. Pokud ale byla k danému klíči nalezena v hashmapě položka, zapíšou se potomci aktuálního uzlu do `ArrayListu` v daném umístění. Celý tento blok se opakuje pro všechny potomky nejvyššího uzlu (`tempEntry`).

Pokud již byly prohledány všechny elementy na výskyty hodnot a hashmapa není prázdná, pro každý klíč z hashmapy se načte jeho `ArrayList` obsahující elementy daného klíče. Tyto elementy jsou poté tisknuty na výstup mezi uměle vytvořený počáteční a koncový element s názvem stejným, jako je klíč hashmapy. Tímto je zajištěno předání všech elementů na výstup do správné struktury. Může ale nastat situace, že je hashmapa prázdná, to znamená, že na hledané hodnotové cestě se nenachází žádný element. V tomto případě jsou všichni potomci objektu `tempEntry` vytištěni na výstup beze změn.

7.3 Objekt tempEntry

`tempEntry` je objekt typu `MainEntry` (viz kapitola 7.1), do kterého se při shodě hledané transformační cesty a aktuálního umístění v dokumentu ukládají načítané struktury elementů. V těchto strukturách poté hledáme vyskytující se hodnoty elementů, se kterými pracujeme při modifikaci struktury výsledného dokumentu. V praxi to funguje tak, že z hledané cesty je vybrán podřetězec cesty končící rodičem hledané cesty.

Pokud tedy chceme seřadit knihy v knihovně podle roku vydání, hledanou hodnotovou cestou bude například sekvence `/books/book/year`. Pro tuto cestu určíme transformační cestu jako `/books/book`. Při procházení dokumentu a načtení počátečního tagu tohoto uzlu `/books/book` se vytvoří instance `tempEntry` a aktuální uzel `/books/book` je přidán jako potomek objektu `tempEntry`. Potomci tohoto uzlu jsou načítáni do seznamu potomků. Při hledání požadované hodnoty se poté tento seznam potomků prochází a zjišťují se jejich hodnoty (viz kapitola 7.2 strana 37).



Obrázek 11: Aktivitní diagram procesu transformace dokumentu

8 Testování

V této testovací části se seznámíme s námi prováděnými druhy testování a jejich účely. Námi vytvořený program je implementován pro analýzu a transformaci XML dokumentu. Při těchto úkonech program zobrazuje i časovou náročnost jednotlivých fází programu.

V první části testů je tato časová náročnost porovnávána s časovou náročností XSLT procesoru Saxon (viz kapitola 3.4). Tyto dva programy jsou schopny transformovat XML dokument se stejným cílem. Liší se ovšem způsoby, kterými těchto cílů dosahují. Námi vytvořený program je speciálně navržen pro tento typ transformace, Saxon je univerzálním XSLT procesorem a obsahuje mnoho vnitřních metod pro rozbor struktury XML dokumentu. Budeme tedy sledovat časovou a paměťovou náročnost obou programů při transformacích podle různých hodnotových cest v dokumentu. Ze získaných hodnot poté utvoříme grafické znázornění pro porovnání. Pro tento typ testů je třeba utvořit příslušnou XSLT šablonu pro procesor Saxon, tomuto návrhu se budeme věnovat v další kapitole.

V druhé části testů budeme testovat účinnost transformací pro původní a transformovaný XML dokument. Použijeme k tomuto testování kolekci knihoven Xerces, které jsou určeny pro parsování, validaci, serializaci a manipulaci s XML. Tyto knihovny implementují API pro parsování XML dokumentů pomocí DOM a SAX (viz kapitola 3.1). V těchto testech se budeme zaměřovat především na celkový čas dotazu a počet diskových přístupů.

8.1 Tvorba XSLT šablony

Abych mohl porovnávat rychlost a účinnost transformace pomocí procesoru Saxon, musel jsem vytvořit XSLT šablonu, která musí obsahovat stejnou funkčnost, jakou vykonává program v transformační části. Toho jsem docílil pomocí funkce `<xsl:for-each-group>`. Tato funkce zajistí rozdělení podstruktury aktuálního elementu podle hodnot jednoho z podelementů. Například mějme v XML souboru strukturu knihovy obsahující názvy žánrů jako „Beletrie“, „Drama“ apod. a v každém tomto žánru jsou elementy knih obsahující její vlastnosti například „Název“, „Autor“, „Rok vydání “. Pokud budeme chtít roztřídit knihy v Beletrii podle autora, zapíšeme část XSLT šablony následovně:

```
1  <!-- Setrideni polozek podle hodnoty elementu -->
2  <xsl:for-each-group select="kniha" group-by="autor">
3      <!-- Pro kazdou skupinu hodnot elementu se vytvori umely element -->
4      <xsl:element name="{current-grouping-key()}">
5          <!-- A struktury obsahujici prislusnou hodnotu elementu se kopiruji -->
6          <xsl:for-each select="current-group()">
7              <xsl:copy-of select = "." />
8          </xsl:for-each>
9      </xsl:element>
10 </xsl:for-each-group>
```

Výpis 6: Příklad použití for-each-group

Následující výpis XSLT šablony je rozdělen do několika částí. Na řádcích 5-13 je aplikování šablony na celý XML dokument. Řádky 15-21 se zpracovávají, pokud se nenacházíme v požadovaném elementu pro úpravy (převedeno do předchozího příkladu je požadovaný element Beletrie). Řádky 23-43 se provádějí při transformaci XML dokumentu.

Jako transformační cesta je v tomto případě řetězec `/knihovna/Beletrie/kniha` a hodnotová cesta je řetězec `/knihovna/Beletrie/kniha/autor`.

```
1  <xsl:stylesheet xmlns:xsl = "http :// www.w3.org/1999/XSL/Transform" version = "2.0"
2  xmlns:xs="http :// www.w3.org/2001/XMLSchema">
3  <xsl:output method = "xml" indent = "yes" />
4
5  <!-- Hlavni cast -->
6  <xsl:template match = "/" >
7
8    <!-- Aplikovani sablon na vstupni XML -->
9    <xsl:copy >
10     <xsl:apply-templates />
11   </xsl:copy>
12
13 </xsl:template>
14
15 <!-- V pripade, ze se nejedna o transformacni cestu v dokumentu,
16 kopiruje se aktualni uzel bez uprav -->
17 <xsl:template match = "node()|@*" >
18   <xsl:copy >
19     <xsl:apply-templates select = "node()|@*" />
20   </xsl:copy>
21 </xsl:template>
22
23 <!-- V pripade, ze se jedna o transformacni cestu v dokumentu,
24 upravime strukturu vypisu -->
25 <xsl:template match = "/knihova/Beletrie" >
26
27   <!-- Vytvoreni elementu -->
28   <xsl:element name="Beletrie">
29
30     <!-- Setrideni polozek podle hodnoty elementu -->
31     <xsl:for-each-group select="kniha" group-by="autor">
32
33       <!-- Pro kazdou skupinu hodnot elementu se vytvori umely element -->
34       <xsl:element name="{current-grouping-key()}">
35
36         <!-- A struktury obsahujici prislusnou hodnotu elementu se kopiruji -->
37         <xsl:for-each select="current-group()">
38           <xsl:copy-of select = "." />
39         </xsl:for-each>
40       </xsl:element>
41     </xsl:for-each-group>
42   </xsl:element>
43 </xsl:template>
44
45 </xsl:stylesheet>
```

Výpis 7: XSLT šablona

8.2 Rychlost transformace

Pro testy transformace jsem používal program XMark [15] pro generování XML dokumentů různých velikostí. K testování jsem použil generované dokumenty o velikosti 10MB a 100MB. Statistiky jednotlivých XML dokumentu lze vidět v tabulce pod tímto odstavcem. V testech transformace jsem u jednotlivých transformací podle různých hodnotových cest v dokumentu měřil čas transformace a paměťovou náročnost. Výsledný čas transformace je průměrem tří měření. Paměťová náročnost je hodnota špičky využití paměti procesem - měřeno Task managerem v operačním systému Windows XP. Ze statistických důvodů jsem si také poznamenal údaje o dané cestě.

Soubor	Počet elementů	Počet cest	Průměrný počet sousedů
10 MB	170 000	493	13,6
100 MB	1 780 000	513	121,4

Tabulka 3: Tabulka statistik XML dokumentů

Pro testování na 10MB i 100MB XML souboru jsem zvolil hodnotové cesty uvedeny v tabulce 4. Hodnotové cesty byly zvoleny s ohledem na celkový dopad transformace, proto jsou zde pro porovnání použity i hodnotové cesty ne příliš vyhovující (cesty 1 a 5). Nejoptimálnější volbou jsou hodnotové cesty, jejichž rodič má co nejvíce sousedů a samotná hodnotová cesta má co nejmenší počet různých hodnot. Tímto dosáhneme co nejlepšího poměru mezi počtem nově vytvořených elementů a počtem jejich podelementů.

V následujících tabulkách 5 a 6 můžeme vidět přehled naměřených hodnot času a využité paměti pro transformaci 10MB a 100MB vstupních souborů. K těmto naměřeným hodnotám je pak na obrázcích 12 až 15 toto porovnání znázorněno graficky. Grafy srovnávají zvlášť časovou a paměťovou statistiku pro každý ze zkoumaných souborů, jsou tedy uvedeny celkem 4 grafy.

Název	Transformační cesta	Hodnotová cesta
Cesta 1	/site/people/person/profile/	/site/people/person/profile/gender
Cesta 2	/site/regions/namerica/item/	/site/regions/namerica/item/payment
Cesta 3	/site/regions/namerica/item/	/site/regions/namerica/item/quantity
Cesta 4	/site/regions/europe/item/	/site/regions/europe/item/quantity
Cesta 5	/site/regions/europe/item/	/site/regions/europe/item/location

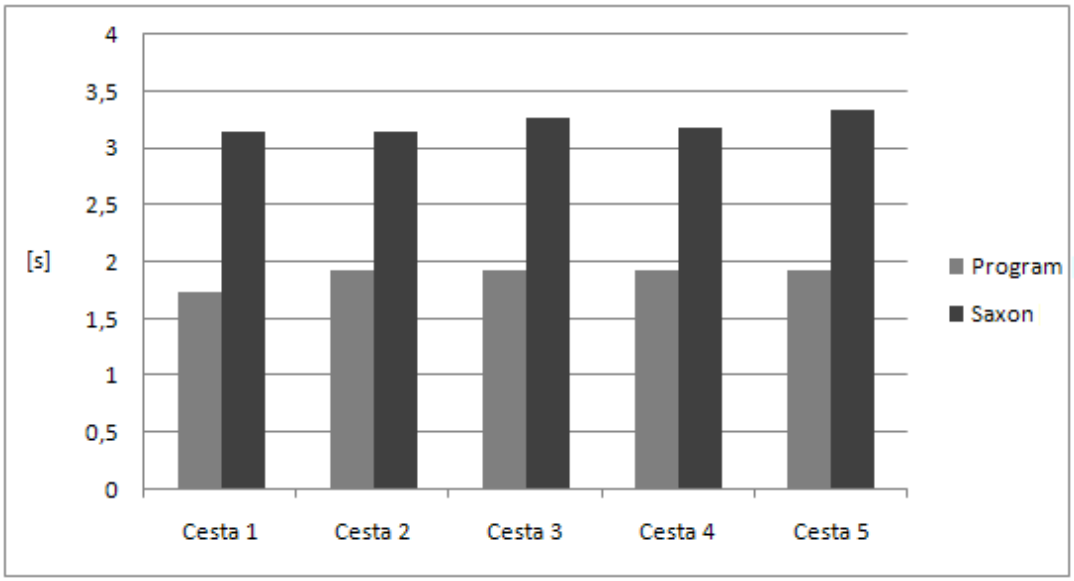
Tabulka 4: Tabulka transformačních a hodnotových cest

	Cesta 1	Cesta 2	Cesta 3	Cesta 4	Cesta 5
Program					
Čas analýzy [min]			5.44		
Paměť. náročnost analýzy [MB]			48528		
Postanalýza [s]	0.657	0.745	0.843	0.735	0.703
Transformace [s]	**1.734**	**1.922**	**1.938**	**1.923**	**1.935**
Paměť [MB]	47048	65319	65548	65608	55512
Saxon					
Transformace [s]	**3.141**	**3.156**	**3.266**	**3.188**	**3.343**
Paměť [MB]	112464	111540	112372	112756	111996
Statistiky elementů					
Vytvořeno	527	15	3	3	102
Ovlivněno	527	807	859	516	516
Poměr	1	53.80	286.33	172	5.06
Rodič sousedů	1	859	859	516	516
Cesta sousedů	1	1	1	1	1
Cesta různých hodnot	2	11	3	3	50

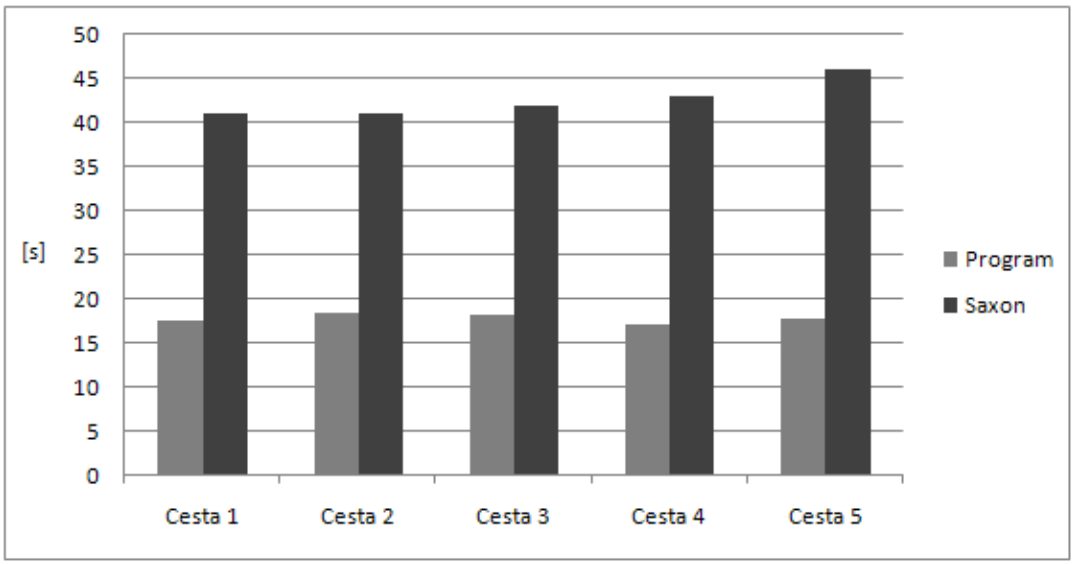
Tabulka 5: Transformace na 10MB souboru

	Cesta 1	Cesta 2	Cesta 3	Cesta 4	Cesta 5
Program					
Čas analýzy [min]			56.5		
Paměť. náročnost analýzy [MB]			49285		
Postanalýza [s]	5.984	7.578	6.936	6.437	6.5
Transformace [s]	**17.672**	**18.391**	**18.334**	**17.156**	**17.735**
Paměť [MB]	55520	188900	173996	129676	130984
Saxon					
Transformace [s]	**41**	**41**	**42**	**43**	**46**
Paměť [MB]	548336	544928	544500	544300	559464
Statistiky elementů					
Vytvořeno	5500	15	5	4	231
Ovlivněno	5500	8118	8680	5208	5208
Poměr	1	541.20	1736	1302	22.55
Rodič sousedů	1	8660	8680	5208	5208
Cesta sousedů	1	1	1	1	1
Cesta různých hodnot	2	11	5	4	50

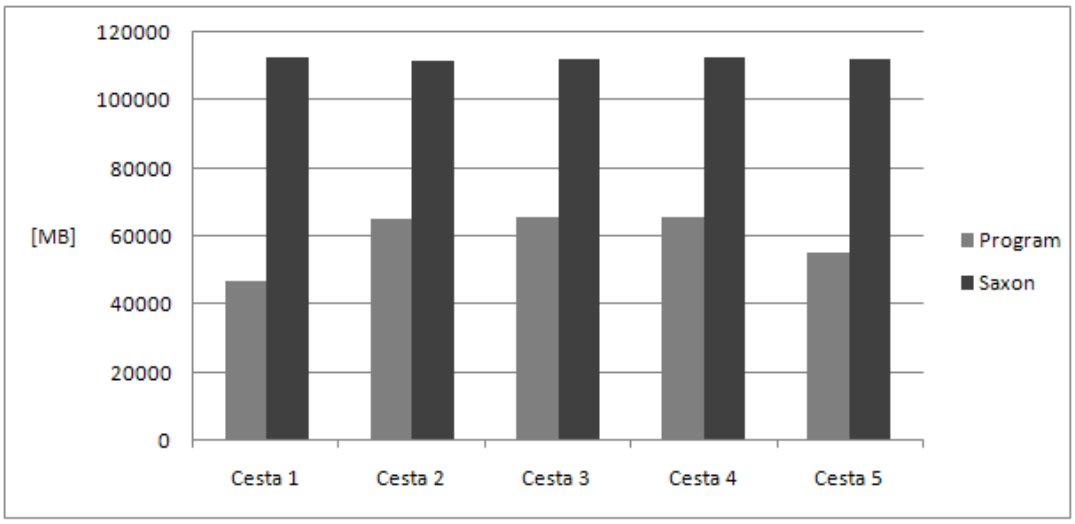
Tabulka 6: Transformace na 100MB souboru



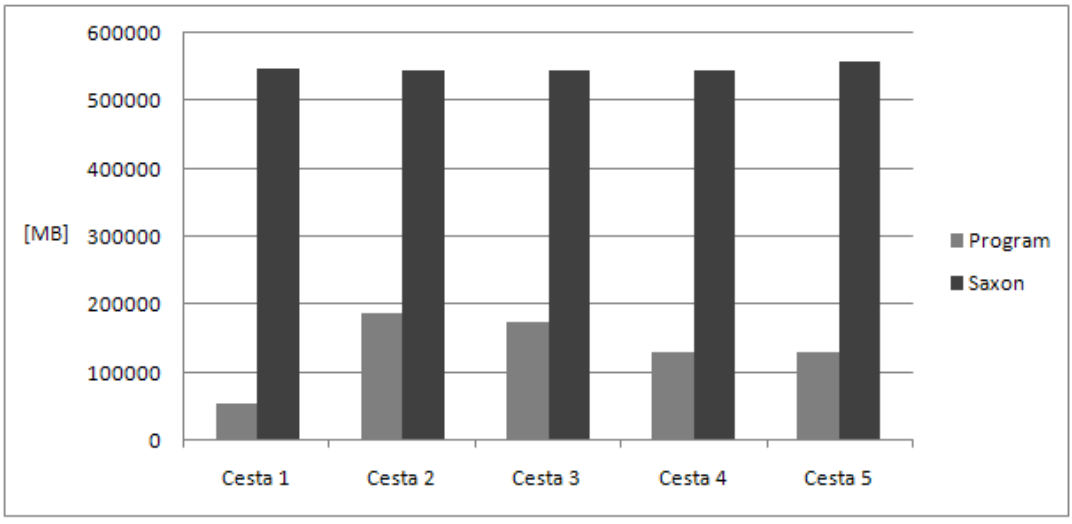
Obrázek 12: Čas trvání transformace pro 10MB soubor



Obrázek 13: Čas trvání transformace pro 100MB soubor



Obrázek 14: Využití paměti při transformaci 10MB souboru



Obrázek 15: Využití paměti při transformaci 100MB souboru

8.3 Účinnost transformace

Pro testování účinnosti jednotlivých transformací jsem používal aplikaci TJDewey [16] využívající kolekci knihoven Xerces. Program funguje na principu indexace XML souboru kdy je poté možno nad těmito indexy spouštět jednoduché XPath dotazy. Program na základě těchto dotazů zobrazuje statistiky hledání dotazu nad indexovanými daty.
Pro testování účinnosti transformace jsem pro každou testovanou hodnotovou cestu, shodnou s cestou v testu transformace (kapitola 8.2), utvářel 2 sady XPath dotazů. Přehled dotazů lze najít v tabulce s výsledky. První dotaz jsem pokaždé aplikoval na originální XML soubor a druhý (upravený pro novou strukturu XML souboru) jsem aplikoval na transformovaný XML soubor pomocí mnou vytvořené aplikace. U výsledků každého dotazu jsem se zajímal o tyto statistiky:

- Query processing time - celkový čas dotazu
- DAC sum - počet diskových přístupů
- Matched solutions - počet objektů vrácených dotazem pro kontrolu správnosti dotazu

Jednotlivé dotazy byly kladeny na odpovídající 10MB a 100MB XML soubory, které byly výstupy z první části testu - testu rychlostí transformace. Cílem těchto testů bylo zjistit, jak mnoho se zlepší vyhledávání dat na optimalizovaném XML souboru. Přehled jednotlivých dotazů lze vidět v tabulce 7 a tabulce 8.

Výsledky těchto měření jsou rozděleny do dvou tabulek, jedna pro 10MB XML soubor a druhá pro 100MB XML soubor. Pro objektivnost hodnot času dotazu jsem daný dotaz spouštěl 6x, odstranil nejlepší a nejhorší čas a zbylé 4 časy zprůměroval. Tento průměr jsem zapsal do tabulky. Hodnoty diskových přístupů byly vždy stejné. Z tabulek jsou poté utvořeny grafy, ve kterých se porovnává rychlost a počet diskových přístupů, zvlášť pro 10MB a 100MB soubor. V těchto grafech lze přehledně vidět, jak moc byla daná transformace účinná.

Název	XPath výraz
Dotaz 1	/site/people/person/profile[/gender='male']
Dotaz 2	/site/people/person/profile[/gender='female']
Dotaz 3	/site/people/person/profile[/gender='male' and /business='Yes']
Dotaz 4	/site/regions/namerica/item[/payment='Cash']
Dotaz 5	/site/regions/namerica/item[/payment='Cash' and /location='United States']
Dotaz 6	/site/regions/namerica/item[/quantity='2']
Dotaz 7	/site/regions/namerica/item[/quantity='2' and /location='United States']
Dotaz 8	/site/regions/europe/item[/quantity='4']
Dotaz 9	/site/regions/europe/item[/quantity='4' and /location='Uganda']
Dotaz 10	/site/regions/europe/item[/location='Samoa']
Dotaz 11	/site/regions/europe/item[/location='Samoa' and /payment='Cash']

Tabulka 7: Tabulka testovaných XPath dotazů pro původní XML soubor

Dotaz	XPath výraz
Dotaz 1	/site/people/person/male/profile[/gender]
Dotaz 2	/site/people/person/female/profile[/gender]
Dotaz 3	/site/people/person/male/profile[/gender and /business='Yes']
Dotaz 4	/site/regions/namerica/Cash/item[/payment]
Dotaz 5	/site/regions/namerica/Cash/item[/payment and /location='United States']
Dotaz 6	/site/regions/namerica/2/item[/quantity]
Dotaz 7	/site/regions/namerica/2/item[/quantity and /location='United States']
Dotaz 8	/site/regions/europe/4/item[/quantity]
Dotaz 9	/site/regions/europe/4/item[/quantity and /location='Uganda']
Dotaz 10	/site/regions/europe/Samoa/item[/location]
Dotaz 11	/site/regions/europe/Samoa/item[/location and /payment='Cash']

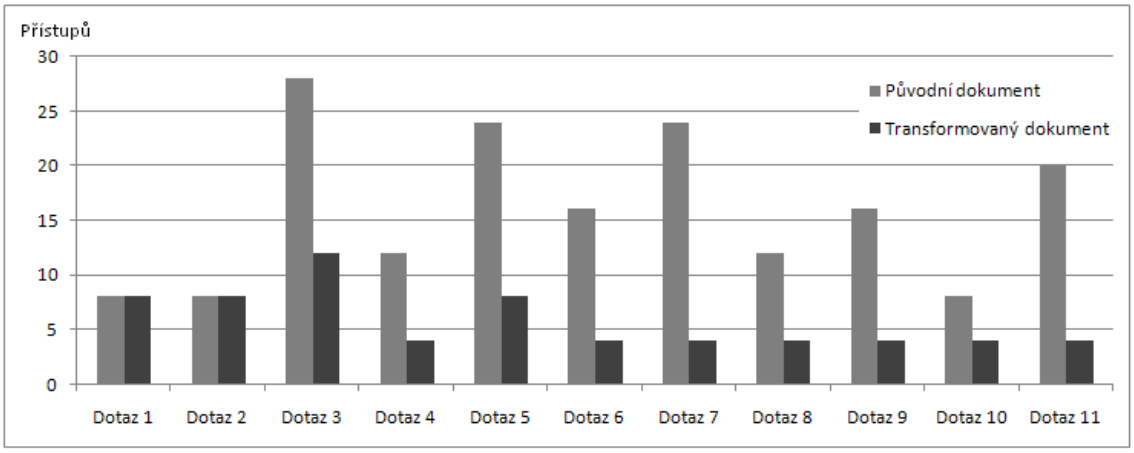
Tabulka 8: Tabulka testovaných XPath dotazů pro transformovaný XML soubor

Dotaz	Celkový čas dotazu [msec]		Počet diskových přístupů	
	Originální	Transformovaný	Originální	Transformovaný
Dotaz 1	31	31	8	8
Dotaz 2	31	15	8	8
Dotaz 3	47	31	28	12
Dotaz 4	31	15	12	4
Dotaz 5	47	31	24	8
Dotaz 6	31	15	16	4
Dotaz 7	31	15	24	4
Dotaz 8	31	31	12	4
Dotaz 9	47	15	16	4
Dotaz 10	47	15	8	4
Dotaz 11	63	15	20	4

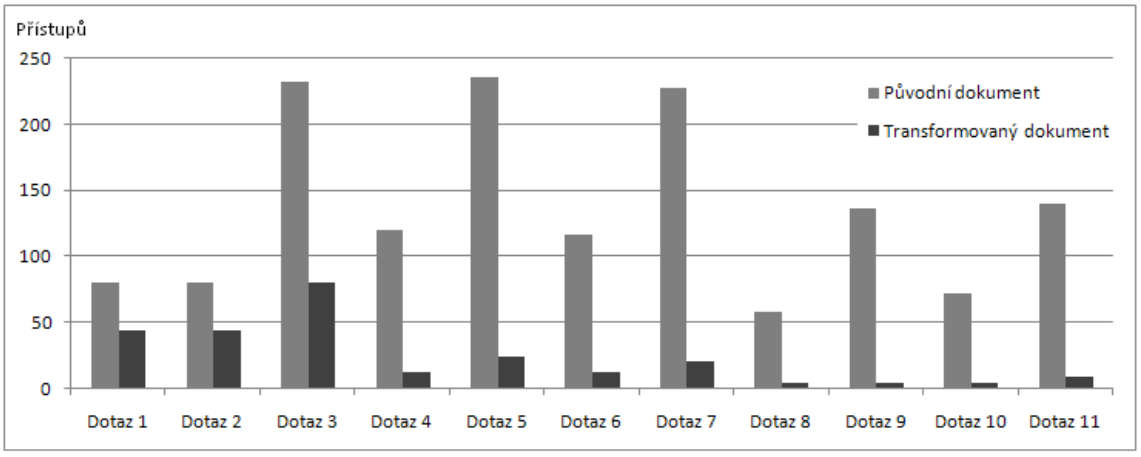
Tabulka 9: Přístupové testy na 10MB souboru

Dotaz	Celkový čas dotazu [msec]		Počet diskových přístupů	
	Originální	Transformovaný	Originální	Transformovaný
Dotaz 1	47	31	80	44
Dotaz 2	47	31	80	44
Dotaz 3	63	47	232	80
Dotaz 4	47	31	120	12
Dotaz 5	63	31	236	24
Dotaz 6	47	31	116	12
Dotaz 7	63	31	228	20
Dotaz 8	63	15	68	4
Dotaz 9	63	31	136	4
Dotaz 10	63	31	72	4
Dotaz 11	63	31	140	8

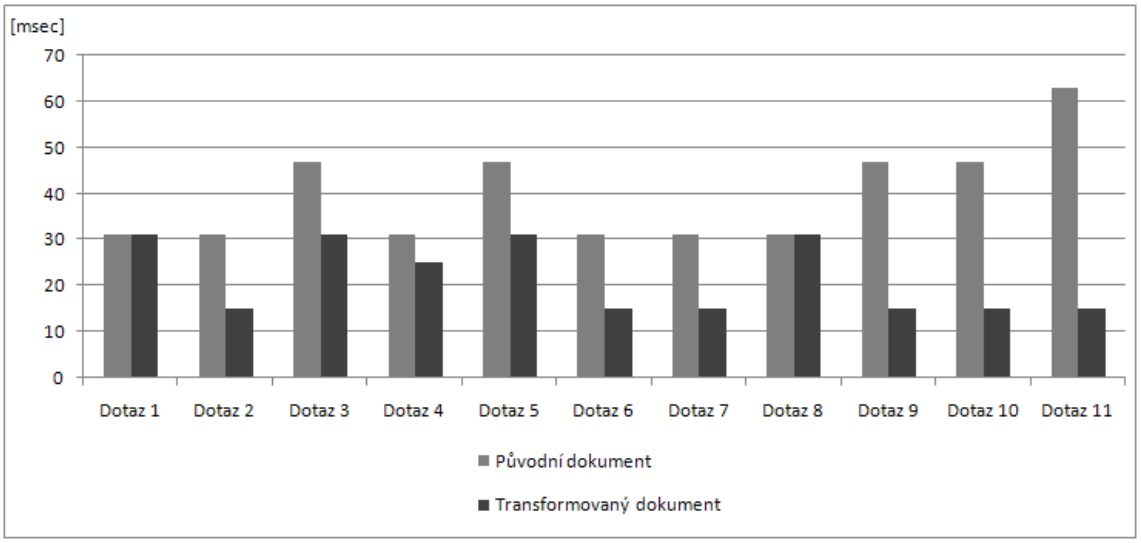
Tabulka 10: Přístupové testy na 100MB souboru



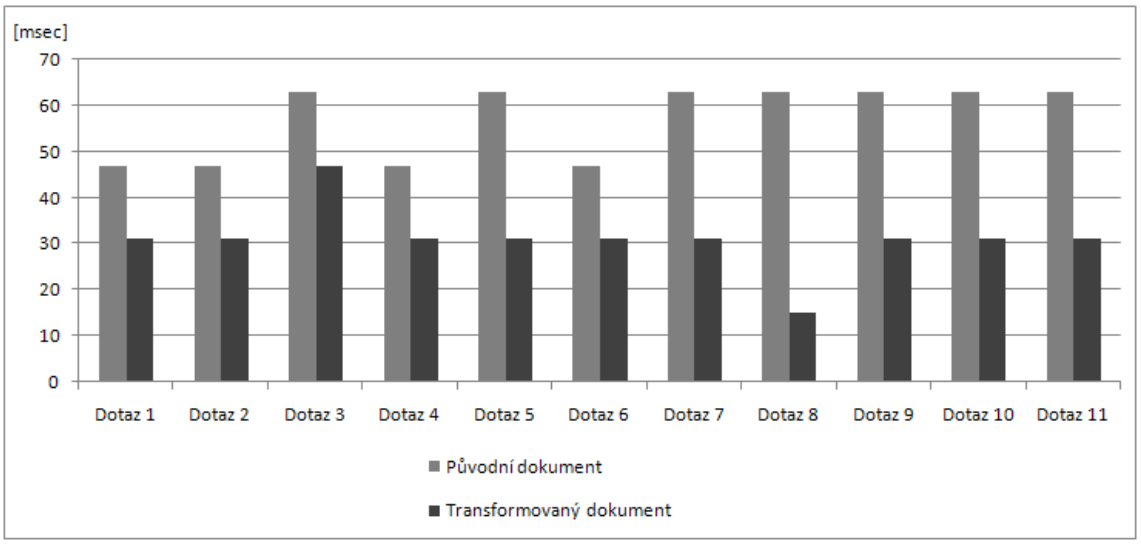
Obrázek 16: Počet diskových přístupů pro 10MB soubor



Obrázek 17: Počet diskových přístupů pro 100MB soubor



Obrázek 18: Výsledné časy dotazů pro 10MB soubor



Obrázek 19: Výsledné časy dotazů pro 100MB soubor

9 Závěr

V této diplomové práci jsme se zabývali problematikou vyhledávání dat v XML souborech. Za účelem optimalizace vyhledávání dat v těchto souborech jsme prozkoumali možnosti jejich vhodného strukturování a navrhli postup, jak tato data třídit. Pro zajištění této vhodné transformace dat jsme také navrhli a implementovali aplikaci, která jako první krok dokázala analyzovat vstupní soubor dat. Z takto získaných statistických dat jsme poté byli schopni rozhodnout o vhodnosti transformace dle určitých hodnotových cest v tomto XML dokumentu. Jako další krok nám poté program transformoval vstupní dokument na výstupní, optimalizovaný, dokument.
Naším cílem poté bylo ověřit, zdali byl výstupní dokument správně transformován a jaká byla úspěšnost této transformace. Brali jsme také ohled na dobu trvání transformace a využití systémových prostředků při transformaci. Správnost transformace jsme ověřovali hledáním dat v původním a transformovaném souboru, kde počet nalezených hodnot musel být stejný. Pro měření úspěšnosti transformace jsme využili nástroj, který nám analyzoval čas potřebný pro hledání testovaných hodnotových cest a počet diskových přístupů, který byl ukazatelem optimalizace vhodného strukturování dat. Pro srovnání rychlosti transformace jsme poté použili nástroj Saxon, který je uzpůsoben k transformacím XML dokumentů. Abychom mohli takto porovnávat, vytvořili jsme pro tento program šablonu se stejnou funkcí jako námi implementovaný program.
Z uvedených měření jsme došli k několika závěrům. Transformace implementovaným programem trvala minimálně o třetinu kratší čas, nežli programem Saxon. V extrémním případě na největším souboru transformace trvala pouze jednu třetinu času, kterou potřeboval Saxon. Tento fakt byl dán tím, že námi implementovaný program byl uzpůsoben pouze pro tuto činnost, kdežto Saxon je řešení pro větší sadu úloh a zpracovává daleko více statistik. K tomuto faktu se také vztahovala jeho větší paměťová náročnost. Zatímco náš program dosahoval na menších souborech pravidelně asi dvou třetinového paměťového využití, na velkých souborech měl jen třetinovou až desetinovou paměťovou náročnost vůči Saxonu. Za pozornost také stojí fakt, že zatímco paměťová náročnost našeho programu výrazněji kolísala podle množství transformovaných dat, paměťová náročnost Saxonu byla pokaždé skoro stejná. Při testování rychlosti vyhledávání dat nad původním a transformovaným dokumentem bylo hledání prakticky vždy rychlejší nad transformovanými daty. U pár ukázkově nevhodných transformací jsme docílili rychlosti stejné, jako nad původním souborem. U počtu diskových přístupů jsme opět dosáhli lepších výsledků, nežli u původního souboru. Čím větší byl soubor a vhodnější transformovaná struktura, tím menšího počtu jsme dosáhli. V průměru byl počet přístupů na menších souborech třetinový, na větších už v některých případech až stokrát menší. Zjistili jsme tedy, že vhodnou optimalizací menších i velikých XML dokumentů lze několikanásobně zkrátit čas pro jejich prohledávání.
Přínosem této práce pro mne byla možnost seznámit se se standardem XML a jeho použitím v dnešních technologiích. Také jsem si rozšířil znalosti dotazovacího jazyka XPath, ale hlavně jsem se obeznámil s tvorbou a použitím XSLT šablon pro transformaci XML dokumentů s využitím jednoho z XSLT procesorů - Saxonu. Zároveň jsem se utvrdil v přesvědčení, že vhodnou optimalizací lze dosáhnout daleko lepších výsledků v mnoha

ohledech informačních technologií.
Další zlepšení programu bych viděl v optimalizaci procesu analýzy XML dokumentu, konkrétně práce s databázovou logikou, kde je znatelné zpomalení aplikace kvůli časté komunikaci s databází. Také bych viděl potenciál v optimalizaci algoritmu pro modifikaci struktury dokumentu, kde by byla možnost lepšího formátování optimalizované struktury dokumentu.

10 Literatura

[1] Mlýnková, Irena. *XML technologie*. Praha: Grada, 2008. ISBN 80-247-2725-7.

[2] Kosek, Jiří. *XML pro každého*. Praha: Grada, 2000. ISBN 80-7169-860-1.

[3] *XSL Transformations* [online]. c1999, [cit. 25.2.2010]. Dostupné z: <http://www.w3.org/TR/xslt>.

[4] *XSLT Reference* [online]. c2000, [cit. 25.2.2010]. Dostupné z: <http://zvon.org/xxl/XSLTreference>.

[5] *XSLT tutorial* [online]. c2010, [cit. 25.2.2010]. Dostupné z: <http://www.w3schools.com/xsl>.

[6] BŘÍZA, Petr. *Základy jazyka XPath* [online]. c2004, [cit. 25.2.2010]. Dostupné z: <http://interval.cz/clanky/zaklady-jazyka-xpath/>.

[7] *Saxon documentation* [online]. c2009, [cit. 13.3.2010]. Dostupné z: <http://www.saxonica.com/documentation>.

[8] WELCH, Andrew. *Kernow plugin for Saxon* [online]. [cit. 13.3.2010]. Dostupné z: <http://kernowforsaxon.sourceforge.net/>.

[9] Herout, Pavel. *Učebnice jazyka Java*. České Budějovice: Kopp, 2004. ISBN 80-7232-115-3.

[10] *JDK 6 Documentation* [online]. c2010, [cit. 13.3.2010]. Dostupné z: <http://java.sun.com/javase/6/docs/>.

[11] *Eclipse.org* [online]. c2010, [cit. 13.3.2010]. Dostupné z: <http://www.eclipse.org/>.

[12] *Cloudgarden Jigloo plugin manual* [online]. c2009, [cit. 13.3.2010]. Dostupné z: <http://www.cloudgarden.com/jigloo/>.

[13] *JAXB Reference Implementation* [online]. c2010, [cit. 13.3.2010]. Dostupné z: <https://jaxb.dev.java.net/>.

[14] *Indexy v MySQL - praktické ukázky* [online]. c2009, [cit. 15.3.2010]. Dostupné z: <http://weboveaplikace.info/2009/03/06/indexy-v-mysql-prakticke-ukazky/>.

[15] *XMark - XML Benchmark project* [online]. c2009, [cit. 15.3.2010]. Dostupné z: <http://www.xml-benchmark.org/>.

[16] *TJDewey - On the Efficient Path Labeling Scheme Holistic Approach* [online]. c2010, [cit. 15.3.2010]. Dostupné z: <http://portal.acm.org/citation.cfm?id=1617252>.

A Obsluha programu

A.1 Podmínky spuštění

Pro spuštění aplikace musí být splněno několik podmínek:

1. funkční MySql databáze + nastavení databáze v `config.xml` souboru
2. `config.xml` soubor se musí nacházet v kořenové složce projektu (na úrovni složky `src`)
3. funkční instalace Javy 1.6

Aplikaci lze spustit několika způsoby:

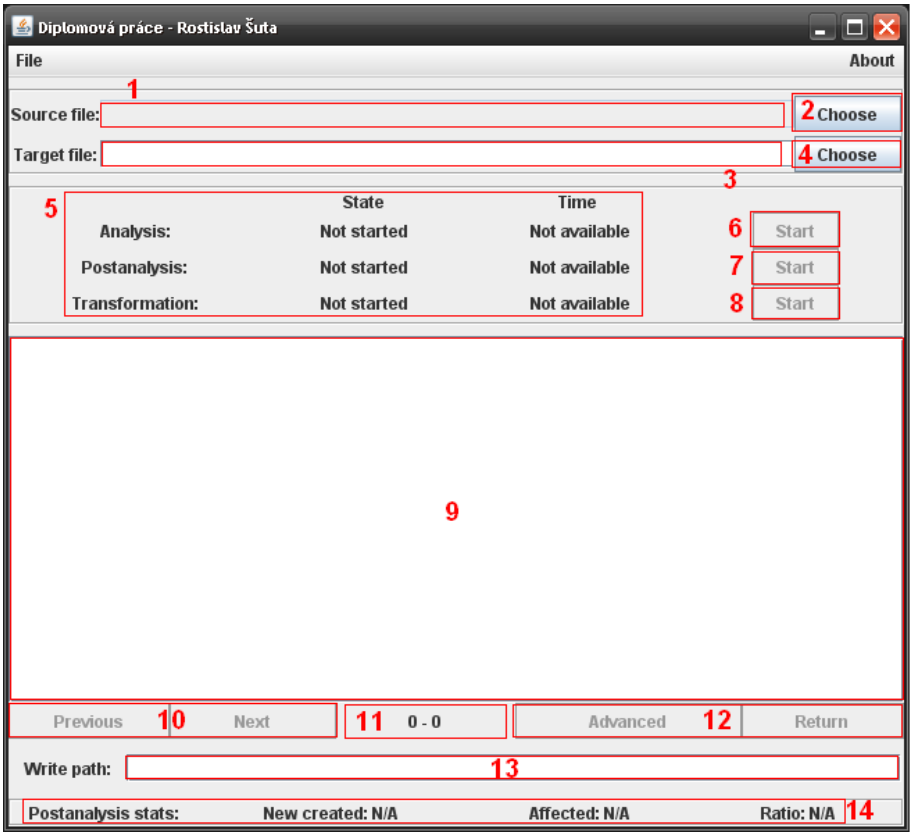
- spuštění připraveným .jar souborem
- spuštění z IDE prostředí (např. Eclipse)

Spuštění připraveným souborem

Program lze jednoduše spustit připraveným .jar souborem. Na soubor stačí klinkout v prostředí Windows, program se automaticky spustí a uživatel může vidět úvodní okno (viz obrázek 20).

Spuštění z IDE prostředí

V IDE prostředí (v našem případě IDE Eclipse) si stačí označit třídu `Gui.java` a zvolit položku „Run as Java application“. Poté by se měla aplikace spustit a měli by jste vidět základní okno aplikace jako na obrázku 20.



Obrázek 20: Rozložení ovládacích prvků

Číslo prvku	Význam
1	cesta ke vstupnímu souboru
2	tlačítko volby vstupního souboru
3	cesta k výstupnímu souboru
4	tlačítko volby výstupního souboru
5	statistiky jednotlivých fází programu
6	spuštění analýzy vstupního souboru
7	spuštění post analýzy vstupního souboru
8	spuštění transformace na výstupní soubor
9	seznam cest v analyzovaném souboru
10	listování v seznamu cest
11	zobrazení pozice v seznamu cest
12	přechod do a ze seznamu potomků zvolené cesty
13	zvolená cesta, podle níž se bude soubor transformovat
14	zobrazení výsledků post analýzy

Tabulka 11: Význam jednotlivých prvků programu

A.2 Konfigurační soubor

Program pro svou funkci a správné nastavení používá konfigurační soubor config.xml uložený v kořenovém adresáři programu. Tento soubor obsahuje nastavení pro práci s databází a XML soubory. Pro názornost uvedu příklad tohoto souboru:

```
<config>
  <database>
    <dbname>transformation</dbname>  // název databáze
    <maintabname>main</maintabname>  // název hlavní tabulky
    <attstabname>atributes</attstabname>  // název tabulky s atributy
    <dburl>localhost:3306/mysql</dburl>    // cesta k databázi
    <dbuser>root</dbuser>  // přihlašovací jméno do databáze
    <dbpassword>password</dbpassword>  // heslo do databáze
  </database>
  <tabnames>
    <id>id</id>      // id záznamu v tabulce
    <name>name</name>  // název záznamu
    <count>count</count>    // počet výskytu záznamů
    <max>max</max>  // maximální počet výskytu záznamu
    <average>avg</average>  // průměrný počet výskytu záznamu
    <different>diff</different>      // počet záznamů s různou hodnotou
    <value>value</value>    // hodnota záznamu
  </tabnames>
  <limits>
    <diffvalues>50</diffvalues>     // limit počtu různých hodnot záznamu pro uchování
    <valuelength>20</valuelength> // limit délky řetězce hodnot záznamu pro uchování
  </limits>
  <files>
    <source>books.xml</source>  // zdrojový XML soubor
    <transformed>transformed.xml</transformed>  // transformovaný XML soubor
  </files>
</config>
```

A.3 Obsluha aplikace

Uživatelské rozhraní

Po spuštění aplikace můžeme vidět okno aplikace jako na obrázku 20. Na obrázku je jednoduše znázorněno rozložení ovládacích prvků aplikace. Pomocí průvodce si popíšeme, jak lze jednoduše postupovat při analýze a transformaci XML dokumentu. V následující tabulce 11 je poté uveden význam jednotlivých prvků.

Analýza

Pro analýzu XML dokumentu musíme zvolit vstupní XML soubor. Dokud soubor nebude vybrán pomocí tlačítka „Choose“ (tlačítko č. 2), nezpřístupní se možnost analýzy souboru (tlačítko č. 6). Až tento dokument zvolíme, můžeme cestu k němu vidět na výpisu (prvek č.1). Výsledky analýzy jsou poté zobrazeny formou seznamu (seznam pod číslem 9).
Při spuštění analýzy je uživatel dotázán, zdali chce využívat již dříve vytvořenou databázi

(její jméno se dá změnit v `config.xml` souboru), nebo zdali chce provést analýzu znovu. Pokud zvolí, že chce využívat již vytvořenou databázi, bude tato databáze načtena a výsledky analýzy budou ihned zobrazeny v seznamu (číslo 9) - analýza se v tomto případě nebude provádět. Pokud bude chtít uživatel provést analýzu znovu, smažou se tabulky v databázi a dojde k nové analýze zdrojového XML souboru. Nové výsledky budou poté zobrazeny v seznamu výsledků (prvek číslo 9).

Seznam výsledků se skládá ze tří sloupců. V prvním sloupci je zobrazen počet sousedů na dané cestě, druhý sloupec zobrazuje počet různých hodnot, kterých daná cesta nabývá a nakonec ve třetím sloupci je vypsaná cesta samotná. V prvním kroku, při výpisu samotných cest jsou výpisy transformačních cest seřazeny podle hodnot v prvním sloupci a to od nejvyšší po nejnižší, při listování seznamem jejich hodnotových cest (viz následující odstavec) jsou pak cesty řazeny dle hodnot ve druhém sloupci a to od nejnižší po nejvyšší. Takto lze přehledně zjistit, do jaké míry je daná cesta vhodná pro transformaci.

V seznamu výsledků lze poté listovat tlačítky „Previous“ a „Next“ (prvky číslo 10). Napravo od těchto tlačítek můžeme vidět, kolik záznamů má aktuální výpis výsledků a ve které části se právě nacházíme. Pokud chceme zobrazit hodnotové cesty náležící aktuální transformační cestě, můžeme tak udělat pomocí tlačítka „Advanced“ a naopak tlačítkem „Return“ se můžeme vrátit ze seznamu hodnotových cest zpět do předchozího výpisu transformačních cest. V poli 13 lze poté vidět, kterou cestu ze seznamu pro transformaci jsme zvolili. Můžeme zde také zapsat cestu vlastní, aniž bychom použili seznam s výsledky.

Post analýza

Pokud jsme již vybrali cestu, podle níž chceme dokument transformovat, můžeme si ověřit vhodnost výsledku post analýzou. K tomuto účelu slouží tlačítko „Start“ (číslo 7). Po stisku tlačítka se provede výpočet, kolik nových elementů bude aplikací vytvořeno a kolik elementů bude touto změnou ovlivněno. Zobrazí se zde také poměr těchto hodnot, který nám udává, jak hodně bude transformace vhodná. Všechny tyto hodnoty lze najít v oblasti číslo 14.

Transformace

Pro zobrazení volby transformace dokumentu (tlačítko číslo 8) musí uživatel zvolit výstupní soubor (tlačítko číslo 4). Poté se při stisku tlačítka „Start“ (číslo 8) provede transformace vstupního dokumentu na výstupní a to podle hodnotové cesty zapsané v poli číslo 13.